**Java 8 New Features**

Java 8 or JDK 1.8 is the most significant expansion of the Java language yet. Java 8’s new features such as Lambda Expressions, Stream APIs, Nashorn, Compact Profiles, new Time APIs increase the expressive power of the platform and make it easier for developers to take advantage of modern, multicore processors. This article gives an overview of the new features in Java 8 with links to in-depth tutorials for the most important of them.

**New Features in Java 8**

1. [**Lambda Expressions**](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as [**Functional Interfaces**](https://www.javabrahman.com/java-8/functional-interfaces-java-8/)) more compactly.
2. [**Method references**](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/) provide easy-to-read lambda expressions for methods that already have a name. [**Constructor References**](https://www.javabrahman.com/java-8/constructor-references-java-8-simplified-tutorial/) are the equivalent forms of representation for constructors.
3. [**Default methods**](https://www.javabrahman.com/java-8/default-methods-in-java-8-with-examples/) enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
4. With default methods in Java 8, [**multiple inheritance of behavior**](https://www.javabrahman.com/java-8/java-8-multiple-inheritance-of-behavior-from-interfaces-using-default-methods/) is now possible in Java, and it is important to understand the [**conflict resolution rules**](https://www.javabrahman.com/java-8/java-8-multiple-inheritance-conflict-resolution-rules-and-diamond-problem/) which resolve *Diamond Problem* and other conflict scenarios.
5. Java 8’s **new package** [**java.util.function**](https://www.javabrahman.com/java-8/java-8-java-util-function-package-tutorial/) provides many useful functional interfaces for the most common scenarios. The 4 most important functional interface among them are – [**Predicate**](https://www.javabrahman.com/java-8/java-8-java-util-function-predicate-tutorial-with-examples/), [**Consumer**](https://www.javabrahman.com/java-8/java-8-java-util-function-consumer-tutorial-with-examples/), [**Function**](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/) and [**Supplier**](https://www.javabrahman.com/java-8/java-8-java-util-function-supplier-tutorial-with-examples/).
6. [**Repeating Annotations**](https://www.javabrahman.com/java-8/java-8-repeating-annotations-tutorial/) provide the ability to apply the same annotation type more than once to the same declaration or type use.
7. New **java.util.stream** package provides a new [**Streams API**](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/) to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API.
8. Java 8’s [**new Collector interface**](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/) and its multiple predefined implementations provide an efficient way to terminate the Stream operations and collect the result in a collection.
9. **A**[**new Date-Time package**](https://www.javabrahman.com/java-8/overview-of-java-8-new-date-time-api-java-time-package-tutorial/)**– java.time** – with a new comprehensive set of date and time utilities.
10. Java 8’s **new enhanced methods in Collections API** are covered in a series of 4 tutorials – [**Iterable.forEach() & Iterator.remove()**](https://www.javabrahman.com/java-8/java-8-iterable-foreach-iterator-remove-methods-tutorial-with-examples/),  [**Collection.removeIf()**](https://www.javabrahman.com/java-8/java-8-collection-removeif-method-tutorial-with-examples/),  [**List.sort() & List.replaceAll()**](https://www.javabrahman.com/java-8/new-features-in-java-8/www.javabrahman.com/java-8/java-8-list-sort-list-replaceall-methods-tutorial-with-examples/),  and new [**multi-value map**](https://www.javabrahman.com/java-8/java-8-maps-computeifabsent-computeifpresent-getordefault-methods-tutorial-with-examples/) methods.
11. Java 8 has introduced [**new internal iterators**](https://www.javabrahman.com/java-8/java-8-internal-iterators-vs-external-iterators/) based on declarative functional programming style.
12. Comparator interface has undergone a major upgrade in Java 8 with new methods leveraging Java 8’s functional programming features, comparator chaining, in-built null handling, and many more such features. [**Java 8 Comparator tutorial**](https://www.javabrahman.com/java-8/the-complete-java-8-comparator-tutorial-with-examples/) covers these new features in depth.
13. **Nashorn Javascript Engine enhanced** to provide a version of javascript which would run within the JVM
14. **Standard Encoding and Decoding Base64**,**Parallel Array Sorting** and **Unsigned Arithematic Support**.
15. **JDBC 4.2** with new features and notably the JDBC-ODBC bridge has been removed.
16. **Concurrency related important changes** which include –
    * Changes to ConcurrentHashMap to support aggregate operations based on the newly added streams facility and lambda expression.
    * Addition of classes to the java.util.concurrent.atomic package to support scalable updatable variables.
    * Support for a common pool in ForkJoinPool.
    * New **StampedLock** class to to provide a capability-based lock with three modes for controlling read/write access
17. **Type Annotations** provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
18. **Improved Type Inference** and **Method Parameter Reflection**.
19. **Compact Profiles** contain predefined subsets of the Java SE platform and enable applications that do not require the entire Platform to be deployed and run on small devices.
20. **Improved internationalization** including support for Unicode 6.2.0, new Calendar and Locale APIs, Adoption of Unicode CLDR Data and the java.locale.providers System Property and Ability to Install a Custom Resource Bundle as an Extension.
21. In case you are interesting in knowing about the **best books on Java 8** then you can read the [**Reviews of 5 Best Books on Java 8**](https://www.javabrahman.com/reviews/5-best-books-learning-java-8-reviewed/).

**Functional Interfaces in Java 8 Explained**

**Overview** – In this tutorial we will be looking at one of the most fundamental features of functional aspects of Java 8 – functional interfaces. We will start by looking at the definition of functional interfaces and the primary purpose for which they have been added to Java 8. This will be followed by understanding the new **@FunctionalInterface** annotation introduced in Java 8 and see an example explaining the definition of a custom functional interface. Lastly, we will look at examples showing the 3 types of functional interfaces which are available at the disposal of a programmer in Java 8.

**What is a Functional Interface**

A functional interface, introduced in Java 8, is an interface which has only a single abstract method. Conversely, if you have *any* interface which has only a single abstract method, then that will effectively be a functional interface. This interface can then be used anywhere where a functional interface is eligible to be used.

**The primary purpose served by Functional Interfaces**

One of the most important uses of Functional Interfaces is that implementations of their abstract method can be passed around as [lambda expressions](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/). By virtue of their ability to pass around functionality(i.e. behavior), **functional interfaces primarily enable *behavior parameterization*.**

**@FunctionalInterface(*java.lang.FunctionalInterface*) annotation**

**@FunctionalInterface** annotation can be used to explicitely specify that a given interface is to be treated as a functional interface. Then the compiler would check and give a compile-time error in case the annotated interface does not satisfy the basic condition of qualifying as a functional interface(that of having a single abstract method).

Please note that **@FunctionalInterface** is an ***informative annotation***, i.e. it is not mandatory to use this annotation for classifying the given interface as a valid functional interface. *Rather*, if one uses this annotation, then the compiler ensures that the interface is not inadvertently changed in such a way that it no longer remains a functional interface

**Example of using the @FunctionalInterface tag**

The code below shows the same **CustomFunctionalInterface** defined above with just the **@FunctionalInterface** annotation added on top of it to demonstrate the use of the annotation –

**Java 8 code showing usage of @FunctionalInterface tag**

|  |
| --- |
| package com.javabrahman.java8;  @FunctionalInterface  public interface CustomFunctionalInterface {      //Single abstract method    public void firstMethod();    } |

**Examples of functional interfaces**

Lets see some examples of functional interfaces to understand them better. I have split the examples into 3 types –

1. **Custom Or User defined Functional Interfaces** – These are **interfaces defined by the user** and have a single abstract method. These may/may not be annotated by **@FunctionalInterface**. Let us see the code for a user-defined functional interface named **CustomFunctionalInterface** below –

**Example of custom Or user-defined Functional Interfaces**

|  |
| --- |
| package com.javabrahman.java8;  public interface CustomFunctionalInterface {      //This is the only abstract method.Hence, this    //interface qualifies as a Functional Interface    public void firstMethod();    } |

1. **Pre-existing functional interfaces in Java prior to Java 8** – These are interfaces which already exist in Java Language Specification and have a single abstract method. Eg. **java.lang.Runnable**, **java.lang.Callable<V>**. Let use see the code used for defining these pre-existing functional interfaces.

**Example of pre-existing functional interfaces in Java**

|  |
| --- |
| //java.lang.Runnable  @FunctionalInterface  public interface Runnable {      public abstract void run();  }  //java.lang.Callable<V>  @FunctionalInterface  public interface Callable<V> {      V call() throws Exception;  } |

1. **Newly defined functional interfaces in Java 8 in java.util.function package** – These are pre-defined Functional Interfaces introduced in Java 8. They are defined with generic types and are re-usable for specific use cases. One such Functional Interface is the **Predicate<T>** interface which is defined as follows –

**Example of new pre-defined functional interfaces in Java 8**

|  |
| --- |
| //java.util.function.Predicate<T>  @FunctionalInterface  public interface Predicate<T> {      boolean test(T t);  } |

**Conclusion**  
In this tutorial we understood functional interfaces, their primary purpose, learnt the new **@FunctionalInterface** annotation introduced in Java 8 and then finally saw examples of various types of functional interfaces available in Java 8.

**Java 8 Lambda Expressions Tutorial with Examples**

Lambda Expressions are Java’s answer to the functional programming concept of closures. *Well… almost!!*Lets deep dive into the concept of lambda expressions now starting with the definition of lambdas. This will be followed by understanding the structure of lambda expressions and their ‘relationship’ with their counterpart functional interfaces. We will then take a look at an example showing how lambdas are used to pass different behaviors as parameters.

**What are Lambda Expressions**

A Lambda Expression ( or just a ***lambda*** for brevity) is a representation of an anonymous function which can be passed around as a parameter thus achieving behavior parameterization. A lambda consists of a list of parameters, a body, a return type and a list of exceptions which can be thrown. I.e. it is very much a function, just anonymous.

An instance of a lambda can be assigned to any [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) whose single abstract method’s definition matches the definition of the lambda. In fact, a lambda is a less verbose way of defining an instance of an interface provided the interface is functional. Since, the definition of a lambda can match the definition of multiple functional interfaces, hence, a lambda instance can be assigned to any of these matching interfaces.

**Understanding Lambda Expressions**

Let us now take a look at an example of a lambda expression. To understand the usage of lambda expressions from the ground up, we will first have to first define a functional interface.

**Defining a functional interface for which lambda will be created**

|  |
| --- |
| package com.javabrahman.java8;  @FunctionalInterface  public interface FirstInterface {   //the abstract method   public void singleMethod(String param);  } |

**Lambda equivalent for the above functional interface would look like** –

(String param)->{System.out.println("My lambda says "+param);};

**Important points to understand regarding the lambda expression defined above-**

1. The element to the left of the arrow(**->**) are the parameters of the lambda. In this case the input parameter is defined as- **String param**.
2. To the right of the arrow(**->**) we have the body of the lambda. Body is where the actual processing within a lambda happens. I.e. the logic of the lambda goes here. The above lambda has a simple logic. It prints the param passed to it appended to a general string value and does not return anything, i.e. *return type is void*(more on returning values later).
3. Parameter – **String param**, and return type – **void**, both closely match the signature of **singleMethod()**in **FirstInterface** defined above. This matching of signatures allows us to assign an instance of the lambda defined above to an instance of the interface FirstInterface as shown below –

FirstInterface instance = (String param) -> {System.out.println("My lambda says "+ param);};

We can now pass this **instance** as a parameter wherever **FirstInterface** is expected.

**Understanding Lambda return types**

Lambda syntax contains 2 variants of return types. This types are based on the contents in lambda following the arrow(->) sign –

**Variant 1.(parameters) -> expression**– In this variant the return type of the lambda expression will be same as the resultant type of the expression

**Variant 2.(parameters) -> {statements;}** – In this variant, there will be no return type(or *void* return type) unless the statements inside the curly braces explicitly return something at the end. In that case the return type will be same as the type of the variable returned.

**Lambda Expressions in Practice**

Having covered the basics of lambda, its now time for us to see how to use lambda expressions in a program.

Let us extend the above example to illustrate the usage of lambdas. We have already defined an instance of the above lambda as –

**FirstInterface instance=(String param)->{System.out.println("My lambda says "+param);};**

Lets say there is a class **FirstInterfacePrinter** which uses an instance of type **FirstInstance** to print a value. The **FirstInterfacePrinter** class is defined as follows –

**FirstInterfacePrinter.java**

|  |
| --- |
| package com.javabrahman.java8;  public class FirstInterfacePrinter {     public void print(FirstInterface firstInterface){    firstInterface.singleMethod("apple");   }     public static void main(String args[]){    FirstInterfacePrinter printer=new FirstInterfacePrinter();    <span class="jb-highlight-lightblue">printer.print((String param) -> {System.out.println("My lambda says "+ param);});</span>   }  } |

**OUTPUT of the above code**

My lambda says apple

Now lets change the printing behavior a bit. Lets say now we want to print a little differently with the following definition of the lambda expression-  
**FirstInterface secondInstance=(String param)->{System.out.println("My lambda's parameter value is "+ param);};**

The main method of FirstInterfacePrinter now looks like this –

**FirstInterfacePrinter's main() method**

|  |
| --- |
| public static void main(String args[]){   FirstInterfacePrinter printer=new FirstInterfacePrinter();   printer.print((String param) -> {System.out.println("My lambda's parameter value is "+ param);});  } |

**OUTPUT of the above code**

My lambda’s parameter value is apple

As you can see the print function of **FirstInterfacePrinter** printed differently based on the lambda passed to it. In other words, using lambda expressions we could create 2 different and anonymous implementations of **FirstInterface** by simply defining the logic as a lambda.

**Lambdas and Behavior Parameterization**

If you observe closely, behavior of printing was passed as a part of the lambda expression to the print function. This is an example of behavior parameterization – passing of behavior as a parameter. In fact, lambdas are the main instruments of enabling behavior parameterization in Java 8.

**Summary**  
In this fundamental tutorial on lambda expressions in Java 8 we understood the definition, structure, and usage of lambda expressions. We also understood how lambdas help us in achieving behavior parameterization.

**Java 8 Function Descriptors Explained**

**Definition** – In Java 8 a Function Descriptor is a term used to describe the signature of the abstract method of a [Functional Interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/). The signature of the abstract method of a Functional Interface is syntactically the same as the signature of the [Lambda Expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/). Hence, a Function Descriptor also describes the signature of a lambda.

**Examples of function descriptors**

To illustrate let us see a few examples of function descriptors –

**Example 1**: Lets say we have a functional interface named FirstInterface. It’s described as below –

**Example 1 - FirstInterface.java**

|  |
| --- |
| package com.javabrahman.java8;  @FunctionalInterface  public interface FirstInterface {    //Single abstract method    public void singleMethod(String param);  } |

For the above interface, named FirstInterface, **the signature of the abstract method OR the function descriptor is (String) -> void**

**Example 2**: Functional Interface SecondInterface.java has a slight variation of the abstract method –

**Example 2 - SecondInterface.java**

|  |
| --- |
| package com.javabrahman.java8;  @FunctionalInterface  public interface SecondInterface {    //Single abstract method    public long computeSum(int num1, int num2);  } |

**For SecondInterface the function descriptor is (int,int) -> long**

**Example 3**: Java’s in-built java.lang.Runnable interface has a single *public void run()* method. **The function descriptor for Runnable interface will be () -> void**

**Example 4**: Lets see an example of a generic type based in-built functional interface named Function<T, R> introduced in Java 8 –

**Example 4 - In-built java.util.function.Function<T, R>**

|  |
| --- |
| @FunctionalInterface  public interface Function<T, R> {      /\*\*       \* Applies this function to the given argument.       \* @param t the function argument       \* @return the function result of type R       \*/      R apply(T t); |

**The function descriptor for Function<T, R> will be T -> R**

**Java 8 Method References Tutorial with Examples**

This tutorial explains the concept of Method References introduced in Java 8. It first defines method references and explains its syntax. Next it looks at the 3 types of method references and explains each of them with code examples.

**Definition:** A method reference is a simplified form (or short-hand) of a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/). It specifies the class name or the instance name followed by the method name. Instead of writing the lambda expression with all the details such as parameter and return type, a method reference lets you create a lambda expression from an existing method implementation.

**Method Reference Syntax**: **<class or instance name>::<methodName>**

**Method Reference Example**

**Integer::parseInt** is a method reference with the following charecteristics –

* It is equivalent to the lambda –

**(String str, Integer integer)->Integer.parseInt(str)**

* It can can be assigned to a [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) **Function<T ,R>** like this –

**Function<String,integer> intParser=Integer::parseInt**

* Above assignment is equivalent to the assignment of lambda expression of **parseInt()** –  
    **Function<String,Integer> intParser =**

**(String str,Integer integer)->Integer.parseInt(str)**

Thus, instead of the longer lambda expression, just its concise method reference can be assigned to a functional interface instance.

**Types of Method References**

**Type 1: Reference to a static method** – If we intend to use a static method of a class then instead of writing the lengthier lambda expresion we can just refer to the method via method references.

**Lambda Syntax**: **(arguments) -> <ClassName>.<staticMethodName>(arguments);**

**Equivalent Method Reference**: **<ClassName> :: <staticMethodName>**

Let us now see an example showing the usage of method reference for a static method –

**Example 1: Reference to a static method**

**package com.javabrahman.java8;**

**import java.util.Arrays;**

**import java.util.List;**

**import java.util.function.BiPredicate;**

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

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

**public class MethodReferenceExample {**

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

**Function<String, Double> doubleConvertor=Double::parseDouble;**

**Function<String, Double> doubleConvertorLambda=(String s) -> Double.parseDouble(s);**

**System.out.println("double value using method reference - "+ doubleConvertor.apply("0.254"));**

**System.out.println("double value using Lambda - "+ doubleConvertorLambda.apply("0.254")); }**

**}**

**}**

**OUTPUT of the above code**

double value using method reference – 0.254  
double value using Lambda – 0.254

**Explanation of the output**

The lambda and method reference worked the same and printed the same double value for the same input passed.

**Type 2: Reference to an instance method of a particular object** –

**Lambda Syntax**: **(param, rest of params)-> (param).<instanceMethodName>(rest of params)**

**Equivalent Method Reference**: **<ClassName> :: <staticMethodName>**

*Note*: **<ClassName>** in method reference is the class of parameter named **param**.  
**Code Example**:*(Skeleton code being the same Type 1 – only giving the delta code below)*

**Example 2:Reference to an instance method of an object**

**Consumer<String> stringPrinter=System.out::println;**

**Consumer<String> stringPrinterLambda=(String s) -> System.out.println(s);**

**stringPrinter.accept("Print from method reference based instance");**

**stringPrinterLambda.accept("Print from instance created from Lambda");**

**OUTPUT of the above code**

Print from method reference based instance  
Print from instance created from Lambda

**Explanation of the output**

The lambda and method reference worked the same and printed the string value passed to them.

**Type 3: Reference to an instance method of an arbitrary object of a particular type**– Here the method reference used is of an instance method of an existing object.

**Lambda Syntax**: **(arguments) -> <expression>.<instanceMethodName>(arguments)**

**Equivalent Method Reference**: **<expression> :: <instanceMethodName>**

**Code Example***(Skeleton code being the same Type 1 – only giving the delta code below)*

**Example 3:Reference to an instance method of an arbitrary object of a particular type**

**List<Integer> intList=Arrays.asList(1,2,3,4);**

**BiPredicate<List>Integer>,Integer> isPartOf=List::contains;**

**BiPredicate<List<Integer>,Integer> isPartOfLambda=(List<Integer> listInt, Integer value) ->**

**listInt.contains(value);**

**System.out.println("Is 1 a part of the intList - "+ isPartOf.test(intList, 1));**

**System.out.println("Is 1 a part of the intList - "+ isPartOfLambda.test(intList, 1));**

**OUTPUT of the above code**

Is 1 a part of the intList – true

Is 1 a part of the intList – true

**Explanation of the output**

The lambda and method reference worked the same and printed the same boolean value ‘true’ for the same input passed.

Note – **There is a 4th type of specialized method reference called Constructor Reference**. I have written a separate article explaining [Constructor References](https://www.javabrahman.com/java-8/constructor-references-java-8-simplified-tutorial/).

**Summary**  
The above tutorial explained java 8 method references including their definition, syntax and the 3 types of method references with detailed code examples.

**Constructor References Java 8 Simplified Tutorial with examples**

This tutorial explains the new Java 8 feature known as constructor reference. It starts off with explaining what is a constructor reference by showing its structure and an example. Next, the tutorial shows an example scenario where constructor references can be applied for instantiating objects from an object factory.  
(*Note* – If you are new to the concept of method references then I would recommend that first refer the [method references tutorial](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/).)

**What are Constructor References:** Constructor Introduced in Java 8, constructor references are specialized form of method references which refer to the constructors of a class. Constructor References can be created using the **Class Name** and the keyword **new** with the following syntax –

**Syntax of Constructor References**: **<ClassName>::new**

**Constructor Reference Example:** If you want reference to the constructor of wrapper class **Integer**, then you can write something like this –

**Supplier<Integer> integerSupplier = Integer::new**

**Example usage of Constructor References in code**

**STEP 1** – Let us define an **Employee** class with a constructor having 2 parameters as shown below –

**Employee.java**

**public class Employee{**

**String name;**

**Integer age;**

**//Contructor of employee**

**public Employee(String name, Integer age){**

**this.name=name;**

**this.age=age;**

**}**

**}**

**STEP 2**– Now lets create a factory interface for employees

called **EmployeeFactory**. **getEmployee()** method of **EmployeeFactory** will return an employee instance as per the normal design of [factory pattern](https://www.javabrahman.com/design-patterns/factory-method-design-pattern-in-java/).

**Interface EmployeeFactory.java**

**public Interface EmployeeFactory{**

**public abstract Employee getEmployee(String name, Integer age);**

**}**

*Note*– Since **EmployeeFactory** interface has a single abstract method hence it is a [Functional Interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/).

**STEP 3** – The client side code to invoke **EmployeeFactory** to create **Employee** instances would be as follows –

**Client-side code for invoking Factory interface**

**EmployeeFactory empFactory=Employee::new;**

**Employee emp= empFactory.getEmployee("John Hammond", 25);**

**Explanation of the code**

* The Constructor Reference of **Employee**is assigned to an instance of **EmployeeFactory**called **empFactory**. This is possible because the [function descriptor](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/) of **Employee**constructor is same as that of the abstract method of the Functional Interface **EmployeeFactory**i.e.

**(String, Integer) ->Employee.**

* Then the **getEmployee**method of **empFactory**is called with John’s name and age which internally calls the constructor of **Employee**and a new **Employee**instance **emp**is created.

**Summary**: In the above tutorial we understood constructor references by first understanding their definition/structure. Next we saw examples for defining a constructor reference for actual usage and then saw an example practical scenario where constructor references can be used.

**Java 8 java.util.function package tutorial**

**Introduction**: This tutorial explains the newly introduced package in Java 8 *java.util.function*. It first explains the purpose of this package, followed by the contexts in which the functional interfaces defined in this package can be used and finally gives an overview of the most important functional interfaces that this package contains.

*Note – Before reading this tutorial it is important that the reader understands the concepts of functional interfaces(*[*tutorial here*](https://www.javabrahman.com/java-8/functional-interfaces-java-8/)*) and lambda expressions(*[*tutorial here*](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/)*).*

**Purpose of java.util.function package**: For common use cases where a lambda expression or a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/) is needed, these are generally assigned to a *target type* of a Functional Interface which has its [function descriptor](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/)( abstract method’s signature in terms of parameter & return types) which should match the signature of the lambda expression.

**java.util.function package provides a set of re-usable common functional interfaces( and their corresponding lambda) definitions which can be used by the programmer in his code instead of creating brand new functional interfaces.**

For example, when we need to check for a condition and return a boolean value the function descriptor would be **(T)->boolean** where **T** is the parameter to the abstract method/lambda and **boolean** is the return value. Now, wherever we need to use a lambda with the **(T)->boolean** descriptor, we can use the in-built **java.util.function.Predicate<T>** functional interface because the descriptor for **Predicate<T>**’s function is also **(T)->boolean**. Then in our code we can write a method which accepts this functional interface like this –

**public void doSomething(Object someObject,Predicate<T> predicateInstance){**

**//doSomething method's code goes here**

**}**

Where –

* **doSomething()** method uses the instance of **Predicate<T>** for some calculations on **someObject**.
* This **someObject** will be a variable or a collection of type T so that this **predicateInstance**which is also of type T can be applied to **someObject**. By applied I mean that some attribute of T can be tested and a boolean value returned as per the descriptor – **(T)->boolean**.

And on the calling side the method **doSomething()** will be called like this –

**doSomething(someObjectInstance, Integer i-> i>10)**

Where –

* The first parameter is of type **someObject**
* Second parameter is a lambda expression
* The lambda of second parameter has the same *function descriptor* as that of the **Predicate<T>** functional interface and hence can be passed as the second parameter

In the above example, a functional interface called **Predicate<T>**, which is in-built in **java.util.function**package, has been used by passing a lambda expression(**Integer i-> i >10**) which matches the function descriptor of the **Predicate<T>** interface. Similarly, method references, whose method signature matches a specific functional interface’s function descriptor, can also be passed wherever expected target type is of that specific functional interface.

Like the above scenario where a functional interface with descriptor ‘**(T)->boolean**’ was expected, there are various other *standard* functional interface definitions required by programmers in their day-to-day coding, apart from their usage in java libraries. For all such common functional interface requirements Java has provided a set of commonly used functional interfaces through **java.util.function** package.

To sum up this section – **java.util.function** package provides standard library-based functional interface for common requirements as an alternative to creating brand new functional interfaces every time one is needed.

Contexts in which Functional Interfaces can be assigned/used: There are 3 ways\contexts in which functional interfaces are used as target types in code

* **Assignment context** – This refers to cases where a lambda/method reference is assigned to the functional interface **Predicate<T>**’s instance –

**Predicate predicateToTest=(Integer integer -> integer > 10);**

* **Method invocation context** – This is when a lambda/method reference is passed to a method parameter which accepts the equivalent functional interface’s instance –

**stream.filter(String::isEmpty)**

* **Cast context** – where a lambda/method reference of one type is explicitly typecast to a functional interface of another type –

**stream.map((ToIntFunction) e -> e.getSize())**

**4 fundamental and most commonly used functional interfaces**.

*Note – Each Functional Interface name in the table below links to individual tutorial for that interface.*

| **Functional Interface** | **Purpose** |
| --- | --- |
| [Consumer<T>](https://www.javabrahman.com/java-8/java-8-java-util-function-consumer-tutorial-with-examples/) | Represents an operation that accepts a single input argument and returns no result. |
| [Function <T, R>](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/) | Represents a function that accepts one argument and produces a result. |
| [Predicate<T>](https://www.javabrahman.com/java-8/java-8-java-util-function-predicate-tutorial-with-examples/) | Represents a predicate (boolean-valued function) of one argument. |
| [Supplier<T>](https://www.javabrahman.com/java-8/java-8-java-util-function-supplier-tutorial-with-examples/) | Represents a supplier of results. |

**Java 8 java.util.function.Predicate tutorial with examples**

This tutorial explains the functional interface Predicate which has been newly introduced in the java.util.function package. It describes Predicate’s usage with the help of multiple examples.

**What is java.util.function.Predicate** – Predicate is a new [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) defined in java.util.function package which can be used in all the contexts where an object needs to be evaluated for a given test condition and a boolean value needs to be returned based on whether the condition was successfully met or not.  
Since Predicate is a functional interface, hence it can be used as the assignment target for a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/)or a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/).

**Advantage of predefined java.util.function.Predicate** – Wherever an object needs to be evaluated and a boolean value needs to be returned( or a boolean-valued Predicate exists – *in mathematical terms*) the Predicate functional interface can be used. The user need not define his/her own predicate-type functional interface.

**java.util.function.Predicate source**

**java.util.function.Predicate source code**

|  |
| --- |
| package java.util.function;  import java.util.Objects;  @FunctionalInterface  public interface Predicate<T> {   boolean test(T t);  //rest of the code goes here  } |

Where –

* **boolean test(T t)** is the abstract method which will define the signature of the lambda expression/method reference which can be assigned to a target of type **Predicate**.
* T is the type of input to the predicate
* **boolean test(T t)** returns true if the input argument matches the predicate(the test condition), otherwise returns false
* There are 3 default methods & 1 static method in Predicate which I will explain in below sections of this tutorial.

**Example of using Predicate for a boolean condition check** –

**Code showing Predicate being used for a boolean condition check**

|  |
| --- |
| import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  import java.util.function.Predicate;  public class PredicateFunctionExample{   public static void main(String args[]){    Predicate<Integer> positive = i -> i > 0;    List<Integer> integerList = Arrays.asList(                        new Integer(1),new Integer(10),                        new Integer(200),new Integer(101),                        new Integer(-10),new Integer(0));    List<Integer> filteredList = filterList(integerList, positive);    filteredList.forEach(System.out::println);   }   public static List<Integer> filterList(List<Integer> listOfIntegers, Predicate<Integer> predicate){    List<Integer> filteredList = new ArrayList<Integer>();    for(Integer integer:listOfIntegers){     if(predicate.test(integer)){      filteredList.add(integer);     }    }    return filteredList;   }  } |

**OUTPUT of the above code**

1  10  200  101

**Explanation of the code and output**

* The static method **filterList()** takes two inputs –
  + A List of Integers which need to be filtered based on some *condition*
  + An instance of Predicate interface which is the condition for evaluation of each integer passed in the Integer list.
* The **filterList()** method loops through the whole IntegerList and whichever integer passes the condition *test* it is added to the resultant list called **filteredList**.
* The caller of **filterList()** gets a *filtered* list back which contains all the Integers which satisfy the *test condition* i.e. greater then zero or *positive*. This is exactly the output we saw above, i.e. **1 10 200 101**
* A lambda expression **i -> i > 0** assigned to an instance of type Predicate named *positive*.
* This predicate instance **positive** is then passed as the second argument of **filterList()** method
* The input **Integer** list can thus be filtered in a different way by writing a *new*lambda with a new test condition(such as less than zero, greater than 100 etc) and the **filterList()** method will apply the test condition to the input list passed. Thus, the test condition is passed as a parameter to the **filterList()**method using the **Predicate** interface

**Default methods in java.util.function.Predicate**: There a few default methods also provided in the Predicate functional interface which enable us to do various types of boolean operations such as and, or, not(negate) with different instances of Predicate. These default methods are –

| **Default Method Name** | **Explanation** |
| --- | --- |
| **and()** | It does logical AND of the predicate on which it is called with another predicate. Example: **predicate1.and(predicate2)** |
| **or()** | It does logical OR of the predicate on which it is called with another predicate. Example: **predicate1.or(predicate2)** |
| **negate()** | It does boolean negation of the predicate on which it is invoked. Example: **predicate1.negate()** |
| Where, **predicate1** & **predicate2** are instances of Predicate interface/lambda expression/method references | |

Lets say we have the same code as used above for the filterList() method. I.e. filterList will filter the input Integers List based on the *predicate*passed to it.  
What we are going to change is we will pass different predicates ANDed, ORed, Negated using the default methods and see the resultant output in the table below *(Note – I am assuming that all readers are aware of AND/ OR/ NOT operations in boolean logic)*

| **Predicate passed** | **Values in filteredList** | **Explanation** |
| --- | --- | --- |
| Lets say values in input IntegerList are – [-10, 0, 1, 20, 101, 200] | | |
| **(i->i>0).and(i->i>10)** | [20,101,200] | Conditions for i greater than 0 and i greater than 10 leaves us with 3 filtered values 20,100 & 200. |
| **(i->i>0).or(i->i>10)** | [1,20,101,200] | Either i is greater than 0 or i > 10 gives us 4 filtered values 1,20,101 & 200. |
| **(i->i>0).negate()** | [-10,0] | Negation of i greater than zero implies i is less than or equal to 0. This gives us 2 values -10 & 0 |

**Static method isEqual()** – Predicate interface also has a static method **isEqual()** which can be used to compare 2 instances of Predicate functional interface.

Its defined as – **static <T> Predicate<T> isEqual(Object targetRef)**

It returns a predicate that tests if two arguments are equal according to **Objects.equals(Object, Object)**.

**Summary** – We looked at what is java.util.function.Predicate functional interface, what is the advantage of having this interface, we looked at the source including the abstract method, then we looked at the default methods, saw examples of their usages and finally learned about the static method present in the Predicate Interface.

**Java 8 java.util.function.Consumer Tutorial with Examples**

Tutorial explains the in-built functional interface **Consumer<T>** introduced in Java 8. It uses examples to show how the **accept()** & **andThen()** methods of the Consumer interface are to be used.

**What is java.util.function.Consumer**

**Consumer<T>** is an in-built functional interface introduced in Java 8 in the **java.util.function** package. Consumer can be used in all contexts where an object needs to be consumed,i.e. taken as input, and some operation is to be performed on the object without returning any result. Common example of such an operation is *printing* where an object is taken as input to the printing function and the value of the object is printed( we will expand upon the printing example in more detail below when understanding how to use Consumer interface).

Since Consumer is a functional interface, hence it can be used as the assignment target for a lambda expression or a method reference.

In case you are new to functional programming then you might want to go through functional interfaces and lambda expressions tutorials before proceeding ahead – [functional interfaces tutorial](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) , [lambda expressions tutorial](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) .

**Function Descriptor of Consumer<T>**:Consumer’s Function Descriptor is **T -> ()**. This means an object of type T is input to the lambda with no return value. To understand Function Descriptors in details you can refer the [function descriptor tutorial](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/).

**Advantage of predefined java.util.function.Consumer**:In all scenarios where an object is to be taken as input and an operation performed on it, the in-built functional interface **Consumer<T>** can be used without the need to define a new functional interface every time.

**java.util.function.Consumer source code**

|  |
| --- |
| @FunctionalInterface  public interface Consumer<T> {      void accept(T t);      default Consumer<T> andThen(Consumer<? super T> after) {          Objects.requireNonNull(after);          return (T t) -> { accept(t); after.accept(t); };      }  } |

**Salient Points regarding Consumer<T>’s source code**:

* **Consumer**has been defined with the generic type **T** which is the same type which its **accept()** & **andThen()** methods take as input.
* **accept()** method is the primary abstract method of the Consumer functional interface. Its function descriptor being **T -> ()**. I.e. **accept()** method takes as input the type T and returns no value. I will explain usage of **accept()** with detailed example in the next section.
* All lambda definitions for Consumer must be written in accordance with accept method’s signature, and conversely all lambdas with the same signature as that of **accept()** are candidates for assignment to an instance of Consumer interface.
* **andThen()** is a [default method](https://www.javabrahman.com/java-8/default-methods-in-java-8-with-examples/) in Consumer interface. Method **andThen()**, when applied on a Consumer interface, takes as input another instance of Consumer interface and returns as a result a *new*consumer interface which represents *aggregation*of both of the operations defined in the two Consumer interfaces. I will explain the usage of **andThen()** with a detailed example in coming sections.

**Usage of accept() method of Consumer**:

To understand the **accept()** method lets take a look at the example below where I take a list of integers and print them using a method **printList()**. The **printList()** method takes 2 inputs- an instance of Consumer interface which contains the printing logic *and*the list which is to be printed. Lets have a look at the code now, post which I will explain in detail how this code works-

**Code showing usage of Consumer.accept() method**

|  |
| --- |
| package com.javabrahman.java8;  import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  import java.util.function.Consumer;  public class ConsumerFunctionExample{    public static void main(String args[]){      Consumer<Integer> consumer= i-> System.out.print(" "+i);      List<Integer> integerList=Arrays.asList(new Integer(1),                                new Integer(10), new Integer(200),                                new Integer(101), new Integer(-10),                                new Integer(0));      printList(integerList, consumer);   }   public static void printList(List<Integer> listOfIntegers, Consumer<Integer> consumer){    for(Integer integer:listOfIntegers){      consumer.accept(integer);    }   }  } |

**OUTPUT on executing the above code**

 1  10  200  101  -10  0

**Explanation of above example’s Code & Output**

* **ConsumerFunctionExample**is my class with 2 methods – **main()** & **printList()**.
* In **main()** method first an instance of Consumer<Integer> is defined(named **consumer**) using a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) which takes input as an object of type Integer and contains logic to print the value of that Integer.
* Next the **main()** method defines a new list of integers and passes it to the **printList()** method along with the consumer object defined earlier which contains the printing logic.
* The **printList()** method iterates through the list of integers and invokes the **accept()** method of the consumer object for every integer in the list.
* The **accept()** method which works as per the lambda definition assigned to the consumer interface, i.e. **i-> System.out.print(" "+i)** , prints out the value of each integer with a single space character prepended to it. Thus giving the output **1 10 200 101 -10 0** .

**Usage of default method andThen() of Consumer**

To understand the **andThen()** default method of Consumer interface, I have taken the same code as that used for example of **accept()** method above and changed it a bit to show **andThen()** works.

**Code showing usage of default method Consumer.andThen()**

|  |
| --- |
| //import statements are same as in previous example; hence skipped  public class ConsumerFunctionExample{   public static void main(String args[]){    Consumer<Integer> consumer= i-> System.out.print(" "+i);    Consumer<Integer> consumerWithAndThen = consumer.andThen( i-> System.out.print("(printed "+i+")"));    List<Integer> integerList=Arrays.asList(new Integer(1),                              new Integer(10), new Integer(200),                              new Integer(101), new Integer(-10),                              new Integer(0));    printList(integerList,consumerWithAndThen);   }   public static void printList(List<Integer> listOfIntegers, Consumer<Integer> consumer){    for(Integer integer:listOfIntegers){     consumer.accept(integer);    }   }  } |

**Output on executing the above code**

1(printed  1)  10(printed  10)  200(printed  200)  101(printed  101)  -10(printed  -10)  0(printed  0)

**Explanation of the code**

* A new instance of Consumer interface is defined which is assigned a value of the previous consumer interface(which prints just the space pepended integer) *aggregated*with the new lambda (which prints **(printed i)** where i is replaced by the value of the integer passed as input to the lambda).
* We thus have two Consumer Interface based operations aggregated – first one prints the integer value and the second one prints *(print i)* where i is the value of the integer passed.
* The consumer interface instance representing aggregated consumer operations is passed as parameter to the **printList()** method.
* The **printList()** method is still the same as previous example i.e. it simply calls the **accept()** method on the consumer interface passed to it while iterating through the list of integers passed to it.
* While in the previous example the consumer interface passed to it contained a single operation which simply printed the integer value prepended with a single space character; this time the consumer interface does two operations which have been aggregated together for each value – printing the integer value with space prepended and then printing the string (printed i) where is is the integer passed to it. The output of this example is thus –**1(printed  1)  10(printed  10)  200(printed  200)  101(printed  101)  -10(printed  -10)  0(printed  0)**

**Summary**  
In this tutorial we looked at what is the Consumer<T> in-built interface defined in Java 8 and what are its advantages. We then looked at how to use the Consumer interface using its **accept()** method & the **andThen()** default method.

**Java 8 java.util.function.Function Tutorial with Examples**

[December 11, 2015](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/)

Tutorial explains the in-built functional interface **Function<T, R>** introduced in Java 8. It uses examples to show how the **apply()**, **andThen()**, **compose()** & **identity()** methods of the Function interface are to be used.

**What is java.util.function.Function**

**Function<T, R>** is an in-built functional interface introduced in Java 8 in the **java.util.function** package. The primary purpose for which **Function<T, R>** has been created is for mapping scenarios i.e when an object of a type is taken as input and it is converted(or mapped) to another type. Common usage of Function is in streams where-in the map function of a stream accepts an instance of Function to convert the stream of one type to a stream of another type.

Since **Function<T, R>** is a [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/), hence it can be used as the assignment target for a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) or a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/).

**Function Descriptor of Function<T, R>**

**Function<T, R>**’s Function Descriptor is **T -> R**. This means an object of type T is input to the lambda and an object of type R is obtained as return value. To understand Function Descriptors in details you can refer the [function descriptor tutorial](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/).

**Advantage of predefined java.util.function.Function**: In all scenarios where an object of a particular type is the input, an operation is performed on it and and object of another type is returned as output, the in-built functional interface **Function<T, R>** can be used without the need to define a new functional interface every time.

**java.util.function.Function source code**

**java.util.function.Function source code**

|  |
| --- |
| @FunctionalInterface  public interface Function<T, R> {      R apply(T t);      default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {          Objects.requireNonNull(before);          return (V v) -> apply(before.apply(v));      }      default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {          Objects.requireNonNull(after);          return (T t) -> after.apply(apply(t));      }      static <T> Function<T, T> identity() {          return t -> t;      }  } |

**Salient Points regarding Function<T, R>’s source code**

* **Function** interface has been defined with the generic types **T** & **R**, where **T** is the type of the input and **R** is the output type.
* Method**apply()** is the primary abstract functional method of **Function** interface. It takes as input a parameter **t** of type **T** and gives an output object of type **R**.
* **Function<T, R>** has two default methods. First default method **compose()** combines the function on which it is applied(lets call it the *current function*) with another function, named **before**, in such a way that when the combined function is applied then first the **before** function is applied which converts the input type **V** to type **T**. And then the current function converts this object of type **T** to its output type **R**. Thus, the combined function obtained as a result of **compose()** applies both the functions, in the process converting type **V** to **R**.
* The second default method is **andThen()** which combines the function on which it is applied(*current function*) with another function, named **after**, in such a way that when the combined function is called then first the current function is applied which converts the input type **T** to type **R**. And then the **after** function is applied which converts from type **R** to **V**. Thus, the combined function obtained by using **andThen()** default method applies both functions internally, in the process converting type **T** to type **V**.
* **Function<T, R>** also contains a static method **identity()** which is very simple as it returns as-is whatever is given to it as input. In the code above it takes as input a parameter **t** of Type **T** and returns back this **t**.

**Usage of apply() method of Function**

The below code shows example usage of **apply()** method where it converts/maps from a list of Employee types to a list of Strings containing the names of all Employees. Let us now go through the code after which I will explain how the **apply()** method works-

**Java 8 code example showing usage of Function.apply() method**

|  |
| --- |
| package com.javabrahman.java8;  import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  import java.util.function.Function;  public class FunctionTRExample{    public static void main(String args[]){      Function<Employee, String> funcEmpToString= (Employee e)-> {return e.getName();};      List<Employee> employeeList=       Arrays.asList(new Employee("Tom Jones", 45),        new Employee("Harry Major", 25),        new Employee("Ethan Hardy", 65),        new Employee("Nancy Smith", 15),        new Employee("Deborah Sprightly", 29));      List<String> empNameList=convertEmpListToNamesList(employeeList, funcEmpToString);      empNameList.forEach(System.out::println);   }   public static List<String> convertEmpListToNamesList(List<Employee> employeeList, Function<Employee, String> funcEmpToString){     List<String> empNameList=new ArrayList<String>();     for(Employee emp:employeeList){       empNameList.add(funcEmpToString.apply(emp));     }     return empNameList;    }  } |

**OUTPUT of the above code**

Tom Jones

Harry Major

Ethan Hardy

Nancy Smith

Deborah Sprightly

**Explanation of above example’s Code & Output**

* **funcEmpToString**is an instance of **Function<Employee,String>**. This is the **java.util.function.Function** instance which is used to convert/map from an **Employee**object to a **String**value.
* The lambda defining funcEmpToString is – **(Employee e)-> {return e.getName();}** . It takes as input an **Employee** object and returns his\her name, which is a **String**value, as output.
* The list of employees is passed to method **convertEmpListToNamesList()** along with the Function object **funcEmpToString**;
* The method **convertEmpListToNamesList()** iterates over all the employees in the employee list, applies the function **funcEmpToString** to each of the **Employee** objects, getting back the employee names in **String** format, which it puts in a employee name list and sends it back to the **main()** method.
* On printing the employee name list we get the names of all the employees as required.

**Usage of default method andThen() of Function**

As explained earlier, **andThen()** default method combines the current Function instance with another one and returns a combined Function instance which applies the two functions in sequence with the function passed as parameter to **andThen()** being invoked *after* the current function.  
Lets see the example below which uses the same **funcEmpToString** Function as used in the **apply()** method’s usage example above and combines it with a **initialFunction** Function instance which maps\converts a String to its first letter –

**Java 8 code showing usage of default method Function.andThen()**

|  |
| --- |
| //import statements are same as in apply() example  public class FunctionTRAndThenExample{    public static void main(String args[]){      Function<Employee, String> funcEmpToString= (Employee e)-> {return e.getName();};      List<Employee> employeeList=       Arrays.asList(new Employee("Tom Jones", 45),        new Employee("Harry Major", 25),        new Employee("Ethan Hardy", 65),        new Employee("Nancy Smith", 15),        new Employee("Deborah Sprightly", 29));      Function<String,String> initialFunction= (String s)->s.substring(0,1);      List<String> empNameListInitials=convertEmpListToNamesList(employeeList, funcEmpToString.andThen(initialFunction));      empNameListInitials.forEach(str->{System.out.print(" "+str);});   }    public static List<String> convertEmpListToNamesList(List<Employee> employeeList, Function<Employee, String> funcEmpToString){     List<String> empNameList=new ArrayList<String>();     for(Employee emp:employeeList){       empNameList.add(funcEmpToString.apply(emp));     }     return empNameList;    }  } |

**OUTPUT of the above code**

T H E N D

**Explanation of above example’s Code & Output**

* Function instance **funcEmpToString** maps\converts an **Employee** object to a **String** of his\her name.
* Function instance **initialFunction** maps\converts a **String** to its initial or first letter.
* Default method **andThen()** is used to combine **initialFunction** with **funcEmpToString**. What the combined method does is that it first maps an **Employee** to his\her name and then takes out the first letter from the name as a **String** value. This combined function is passed as **Function** parameter to **convertEmpListToNamesList()** method along with the employee list.
* When the **convertEmpListToNamesList()** applies the combined function to each of the **Employee**objects, then the result is a **String** list first letters of names of each employee.
* This is the required output i.e. T H E N D

**Usage of default method compose() of Function**  
As explained earlier, **compose()** default method combines the current Function instance with another one and returns a combined Function instance which applies the two functions in sequence with the parameter function to **compose()** being invoked *before* the current function.  
Lets see the example below which uses the same **funcEmpToString** Function as used in the **apply()** usage example and combines it with a **funcEmpFirstName** Function instance which converts the full-name of the employee object passed to it to just the first name of the employee-

**Java 8 code showing usage of default method Function.compose()**

|  |
| --- |
| //import statements are same as in apply() example  public class FunctionTRComposeExample{    public static void main(String args[]){      Function<Employee, String> funcEmpToString= (Employee e)-> {return e.getName();};      Function<Employee, Employee> funcEmpFirstName=       (Employee e)-> {int index= e.getName().indexOf(" ");                   String firstName=e.getName().substring(0,index);                   e.setName(firstName);                   return e;};      List<Employee> employeeList=        Arrays.asList(new Employee("Tom Jones", 45),         new Employee("Harry Major", 25),         new Employee("Ethan Hardy", 65),         new Employee("Nancy Smith", 15),         new Employee("Deborah Sprightly", 29));      List<String> empFirstNameList= convertEmpListToNamesList(employeeList,funcEmpToString.compose(funcEmpFirstName));      empFirstNameList.forEach(str->{System.out.print(" "+str);});    }   public static List<String> convertEmpListToNamesList(List<Employee> employeeList, Function<Employee, String> funcEmpToString){     List<String> empNameList=new ArrayList<String>();     for(Employee emp:employeeList){       empNameList.add(funcEmpToString.apply(emp));     }     return empNameList;    }  } |

**OUTPUT of the above code**

Tom Harry Ethan Nancy Deborah

**Explanation of above example’s Code & Output**

* Function instance **funcEmpToString** maps\converts an **Employee** object to a **String** value of his\her name.
* Function instance **funcEmpFirstName** maps\converts the **name** inside an **Employee** object to the first name using the **substring** method of **String**.
* Default method **compose()** is used to combine **funcEmpFirstName** with funcEmpToStringString. What the combined method does is that it first converts the name of an **Employee** into just his\her *first name* returning the same **Employee** object back with the changed value of **name**. It then converts\maps the **Employee**object to just its **name** as a **String**.This combined function is passed as **Function<Employee, String>**parameter to **convertEmpListToNamesList()** method along with the employee list.
* When the **convertEmpListToNamesList()** applies the combined function to each of the **Employee**objects, then the result is the list of first names of each employee.
* This is the required output i.e. Tom Harry Ethan Nancy Deborah

**Usage of static method identity() of Function**  
Static method **identity()** is very simple – it just returns back the parameter which it gets as input. Lets see an example to see how **identity()** method works –

**Java 8 code showing usage of static method Function.identity()**

|  |
| --- |
| //import statements are same as in apply() example  public class FunctionTRIdentityExample{    public static void main(String args[]){      Function<Employee, String> funcEmpToString= (Employee e)-> {return e.getName();};      List<Employee> employeeList=       Arrays.asList(new Employee("Tom Jones", 45),        new Employee("Harry Major", 25),        new Employee("Ethan Hardy", 65),        new Employee("Nancy Smith", 15),        new Employee("Deborah Sprightly", 29));      List<Employee> empNameListInitials=applyIdentityToEmpList(employeeList, Function.identity());      empNameListInitials.forEach(System.out::println);   }    public static List<Employee> applyIdentityToEmpList(List<Employee> employeeList, Function<Employee, Employee> funcEmpToEmp){     List<Employee> empNameList=new ArrayList<Employee>();     for(Employee emp:employeeList){       empNameList.add(funcEmpToEmp.apply(emp));     }     return empNameList;    }  } |

**OUTPUT of the above code**

Employee Name:Tom Jones Age:45  
Employee Name:Harry Major Age:25  
Employee Name:Ethan Hardy Age:65  
Employee Name:Nancy Smith Age:15  
Employee Name:Deborah Sprightly Age:29

**Explanation of above example’s Code & Output**

* Function instance **funcEmpToString** maps\converts an **Employee** object to a **String** of his\her name.
* Employee list is passed to the method **applyIdentityToEmpList()** along with an instance of **Function.identity()**. The parameter of **applyIdentityToEmpList()** which takes **Function.identity()** value is nothing but **Function<Employee, Employee>** i.e. an equivalent of a **Function** which takes **Employee** as input and gives back (the same) **Employee** as output.
* Method **applyIdentityToEmpList()** takes the input employee list, iterates through it, applies the **identity()** function to each employee in the list and returns back the list of employees obtained as a result of applying the **identity()** function.
* As we now know, the identity function does *nothing*, it just returns back the object it receives as input. So, what we get back is the same employee list which we passed to the **applyIdentityToEmpList()** method! And the same i.e. original employee list is printed as output!!
* This is how the static method **Function.identity()** works.

**Summary**  
In this tutorial we looked at what is the **Function<T, R>** in-built interface defined in Java 8 and what are its advantages. We then looked at how to use the Function interface using its **apply()** method, default methods **andThen()** & **compose()**, and lastly how to use the static method **identity()**.

**Java 8 java.util.function.Supplier Tutorial with Examples**

[December 16, 2015](https://www.javabrahman.com/java-8/java-8-java-util-function-supplier-tutorial-with-examples/)

**Introduction**: Tutorial explains the in-built functional interface **Supplier<T>** introduced in Java 8. It explains with the help of examples how the Supplier interface is to be used via its **get()** method.

**What is java.util.function.Supplier**: **Supplier<T>** is an in-built [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) introduced in Java 8 in the **java.util.function** package. Supplier can be used in all contexts where there is no input but an output is expected.

Since Supplier is a functional interface, hence it can be used as the assignment target for a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) or a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/).

**Function Descriptor of Supplier<T>**: Supplier’s Function Descriptor is **() -> T**. This means that there is no input in the lambda definition and the return output is an object of type T. To understand Function Descriptors in details you can refer the [function descriptor tutorial](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/).

**Advantage of predefined java.util.function.Supplier**: In all scenarios where there is no input to an operation and it is expected to return an output the in-built functional interface **Supplier<T>** can be used without the need to define a new functional interface every time.

**java.util.function.Supplier source code**

**java.util.function.Supplier source code**

|  |
| --- |
| @FunctionalInterface  public interface Supplier<T> {      /\*\*       \* Gets a result.       \* @return a result       \*/      T get();  } |

**Salient Points regarding Supplier<T>’s source code**:

* **Supplier**has been defined with the generic type **T** which is the same type which its **get()** methods return as output.
* **get()** method is the primary abstract method of the Supplier functional interface. Its function descriptor being **() -> T**. I.e. **get()** method takes no input and returns an output of type T. I will explain usage of **get()**with detailed example in the next section.
* All lambda definitions for Supplier must be written in accordance with **get()** method’s signature, and conversely all lambdas with the same signature as that of **get()** are candidates for assignment to an instance of Supplier interface.

**Usage of get() method of Supplier**:  
To understand the **get()** method lets take a look at the **SupplierFunctionExample**’s code below, post which I have explained in detail how the code works-

**Code showing usage of Supplier.get() method**

|  |
| --- |
| //SupplierFunctionExample.java  import java.util.Date;  import java.util.function.Supplier;  public class SupplierFunctionExample {   public static void main(String args[]) {    //Supplier instance with lambda expression    Supplier<String> helloStrSupplier = () -> new String("Hello");    String helloStr = helloStrSupplier.get();    System.out.println("String in helloStr is->"+helloStr+"<-");      //Supplier instance using method reference to default constructor    Supplier<String> emptyStrSupplier = String::new;    String emptyStr = emptyStrSupplier.get();    System.out.println("String in emptyStr is->"+emptyStr+"<-");      //Supplier instance using method reference to a static method    Supplier<Date> dateSupplier= SupplierFunctionExample::getSystemDate;    Date systemDate = dateSupplier.get();    System.out.println("systemDate->" + systemDate);   }   public static Date getSystemDate() {    return new Date();   }  } |

**OUTPUT of the above code**

String in helloStr is->Hello<-  
String in emptyStr is-><-  
systemDate->Wed Dec 16 19:18:15 IST 2015

**Explanation of above example’s Code & Output**

* **SupplierFunctionExample** is my class with 2 methods – **main()** & **getSystemDate()**.
* **getSystemDate()** is a static method which simply returns the current system date and does not take any input. The method signature matches the function descriptor of Supplier i.e. **() -> T**.
* **In main() method I have shown how to instantiate a Supplier interface instance in following 3 ways**–
  1. **Using a Lambda Expression**: I have defined a a lambda expression which takes no input and returns a new String object with value set to “hello”. This lambda I have assigned to a **Supplier<String>**instance named **helloStrSupplier**. Invoking functional method **get()** on **helloStrSupplier** gives us a String **helloStr** which is then printed to show that it indeed contains the value “hello”.
  2. **Using a Method Reference to default constructor of String**: Method Reference to the default constructor of **String** is used to create a **Supplier<String>** instance named **emptyStrSupplier**. **emptyStrSupplier** is then used to create a String object named **emptyStr** using the **get()** method. **emptyStr**‘s value is then printed to show its value is empty as defined.
  3. **Using a Method Reference to getSystemDate()**: Method Reference to the **getSystemDate()**method of **SupplierFunctionExample** class is used to create a **Supplier<Date>** instance named **dateSupplier**. **dateStrSupplier** is then used to create a **Date** object named **systemDate** by invoking **get()** method on it. **systemDate**‘s value is then printed to show its value. Value printed is of 16-Dec when I ran this example.

**Summary**: In this tutorial we looked at what is the **Supplier<T>** in-built interface defined in Java 8 and what is its main advantage. We then looked at how to use the **Supplier<T>** interface using its **get()** method with an example.

**Java 8 Streams API Tutorial with Examples**

[October 9, 2015](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/)

Java 8 Streams API tutorial starts off with defining Java 8 Streams, followed by an explanation of the important terms making up the Streams definition. We will then look at Java 8 code examples showing how to exactly use Streams API. **By the end of this tutorial you should feel confident of writing your first program utilising Java 8 Streams API.**

**What are Streams/Defining Streams**  
A Stream in Java is a sequence of elements supporting parallel and aggregate operations. This sequence of elements are obtained from a source. Further, streams in Java 8 support processing of these elements through operations defined in a declarative way. These operations are linked-up according to the [principle of pipelines](https://www.javabrahman.com/programming-principles/pipelines-in-computing-and-software-engineering/)and they access the elements via Internal Iterations.

**Conceptual terms making up the Streams definition**  
Streams definition given above has a lot of terms put together. To understand this definition in its totality we need to understand each of these terms. Lets look at them one-by-one:

* **Sequence of elements** – A stream provides an interface to a sequenced set of values of a specific type. For e.g. **Stream<Integer>** is a stream of type **Integer**.
* **Source of stream elements** – Streams are defined as originating from a specific source which can be collections, arrays, input-output(I/O) resources etc
* **Operations on stream elements** – As the stream’s elements are encountered, several pre-defined operations can be declared to act on the stream elements to map, reduce and collect these elements.
* **Parallel and aggregate operations** – The operations working on these stream of elements can work in parallel on multi-core architectures. Aggregate operations act on elements in the stream in a sequence and end up aggregating data into an end value.
* **Pipeline of Operations** – Various operations which have been declared to act on a stream work together based on the concept of Pipelines([link to tutorial](https://www.javabrahman.com/programming-principles/pipelines-in-computing-and-software-engineering/)). I.e. output of one stream operation acts as input of the next stream operation.
* **Internal iterations**– Internal iterations delegate the work of iterating to the Streams library. The programmer just needs to specify in a declarative manner as to which operation has to be applied to the stream of elements.

**Streams API usage example**  
To start with, let us look at a class **Employee.java** which has –

* 2 instance attributes – **name** & **age**.
* Getters and setters for these attributes.
* A constructor with both attributes.
* The **toString()** method.

**Employee.java**

|  |
| --- |
| package com.javabrahman.java8;  public class Employee{    private String name;    private Integer age;    public Employee(String name, Integer age){      this.name=name;      this.age=age;    }    public String getName(){      return name;    }    public void setName(String name){      this.name=name;    }    public Integer getAge(){      return this.age;    }    public void setAge(Integer age){      this.age=age;    }    public String toString(){      return "Employee Name:"+this.name        +"  Age:"+this.age;    }  } |

Next I have a class called **BasicStreams.java**, given below, in which I have –

* A static list of employees called **employeeList**
* **main()** method where I have my streams-based filter logic
* I initialise my list of Employees with 5 Employee Objects using the 2-parameter **Employee** constructor which utilises the **Arrays.asList()** method

**Streams usage example - BasicStreams.java**

|  |
| --- |
| package com.javabrahman.java8;  import java.util.List;  import java.util.Arrays;  import static java.util.stream.Collectors.toList;  public class BasicStreams {   static List<Employee> employeeList=        Arrays.asList(new Employee("Tom Jones", 45),                      new Employee("Harry Major", 25),                      new Employee("Ethan Hardy", 65),                      new Employee("Nancy Smith", 15),                      new Employee("Deborah Sprightly", 29));   public static void main(String args[]){     List<Employee> filteredList = employeeList.stream()                          .limit(2)                          .collect(toList());     filteredList.forEach(System.out::println);   }  } |

**OUTPUT of the above code**

Employee Name:Tom Jones Age:45  
Employee Name:Harry Major Age:25

**Explanation of the code**

* The program starts with static import of **Collector**Class’ static **toList()** method. This method is used to get a list from a stream
* In the **main()** method, **stream()** is invoked on employeeList. The method **stream()** has been newly introduced in Java 8 on the interface Collection which List interface extends. This **stream()** method gives an instance **java.util.Stream** as the return type
* The stream of employees is then passed(or pipe-lined) to the function **limit()**. The **limit()** function puts a limit to the maximum number of elements which will be picked from the stream. In the given example I have passed the value 2, hence the current stream is now limited to first 2 elements. Also, note that **limit()** is an [intermediate operation](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/), i.e. the stream processing does not end with **limit()**method.
* The **collect()** method is then invoked on the stream(which now has only 2 elements). **Collect()** uses a **Collector**of a specific type which in the given example is of type **List**(returned by the static **toList()**method of **Collectors** class). To simplify the previous statements, **collect()** uses the **toList()** method to return a list equivalent of the stream pipe-lined into it(named **filteredList**). Note that **collect()** is a [terminal operation](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/), i.e. the processing of the stream ends with the **collect()** method.
* At the end, I simply use a Java 8 style **forEach**loop and a [Method Reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/) to the**System.out.println()** method to print all the elements in the resultant **filteredList**.
* As expected the 2 employee objects are printed using the overridden **Employee.toString()** method.

**Summary**  
In the above tutorial we saw what Java 8 Streams are, understood the various terms which describe a Stream, and finally saw a basic example of how to start using Streams in your programs using the intermediate & terminal operations.

**Java 8 Filtering and Slicing with Streams Tutorial with examples**

[December 7, 2015](https://www.javabrahman.com/java-8/java-8-filtering-and-slicing-streams-tutorial-with-examples/)

**Introduction**: This tutorial explains how to do filtering and slicing in Java 8 Streams using the filter, distinct, limit and skip methods. It assumes that you are familiar with basics of [Java 8 Streams API](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/).  
**Streams Filtering & Slicing Basics**: Java 8 Streams support declarative filtering out of elements along with the ability to slice-off portions of a list. Streams support four operations to achieve this – **filter()**, **distinct()**, **limit(n)** and **skip(n)**. Lets quickly look at what these methods do followed by a java example which uses all of these.

**Methods in Streams API for filtering and slicing**

1. **Filter method**: The filter method filters out elements from a given stream. It uses a [Predicate function](https://www.javabrahman.com/java-8/java-8-java-util-function-predicate-tutorial-with-examples/)instance which it applies to the whole stream and returns a filtered stream containing those elements which match the Predicate. It is an [intermediate](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/) stream operation.  
   **Syntax of filter method**:**<stream-instance>.filter()**  
   **Returns**: **java.util.Stream**
2. **Distinct Method**: The distinct method when applied to a stream, returns a stream instance which has all elements unique/distinct. I.e. every unique element is present only once in the resultant stream. The *uniqueness* of elements in the resultant stream is determined by the [equals & hashcode](https://www.javabrahman.com/corejava/understanding-equals-and-hashcode-contract-when-using-collections-in-java/) implementation in these elements. It is an intermediate stream operation.  
   **Syntax of distinct method**: **<stream-instance>.distinct()**  
   **Returns**: **java.util.Stream**
3. **Limit(n) method**: The limit method puts an upper-limit cap purely based on the number of elements in the stream. A limit of n applied to a stream returns a stream which contains exactly n elements if the original stream had more than or equal to n elements. In case the original stream had less than n elements, and limit of n has been applied on the stream, then there is no affect on the stream elements, i.e. the returned stream is same as the original stream. This method is a [short-circuiting](https://www.javabrahman.com/programming-principles/short-circuiting-or-short-circuits-in-boolean-evaluations-in-programming-and-java/) intermediate stream operation.  
   **Syntax of limit method**: **<stream-instance>.limit(n)**  
   **Returns**: **java.util.Stream** of n elements
4. **Skip(n) method**: The skip method is complimentary to the limit method. I.e. the skip method returns a truncated version of the original stream such that the first n elements in the list are skipped and the remaining elements are returned in the resulting stream. This method is an intermediate stream operation.  
   **Syntax of skip method**: **<stream-instance>.skip(n)**  
   **Returns**: **java.util.Stream** of elements post first n elements

**The reason limit(n) & skip(n) are complimentary** is because given a stream of size k, streams obtained from **limit(n)** & **skip(n)** applied on the original stream (where, **n<=k**) when aggregated together give us the original stream back. While **limit(n)** gives the first **n** elements from positions **1** to **n**; **skip(n)** gives elements from positions **n+1** to **k**. Thus, joining the streams obtained from **limit(n)** & **skip(n)** gives us the original stream back.

**Java example demonstrating the use of filter, distinct, limit & skip methods**

**Java example demonstrating the use of filter, distinct, limit & skip methods**

|  |
| --- |
| //Employee.java  public class Employee{    private String name;    private Integer age;    public Employee(String name, Integer age){      this.name=name;      this.age=age;    }    public String getName(){      return name;    }    public void setName(String name){      this.name=name;    }    public Integer getAge(){      return this.age;    }    public void setAge(Integer age){      this.age=age;    }    public String toString(){      return "Employee Name:"+this.name        +"  Age:"+this.age;    }    @Override    public boolean equals(Object obj) {       if (obj == this) {         return true;       }       if (!(obj instanceof Employee)) {         return false;       }       Employee empObj = (Employee) obj;         return this.age==empObj.age           && this.name.equalsIgnoreCase(empObj.name);    }    @Override    public int hashCode() {      int hash = 1;      hash = hash \* 17 + this.name.hashCode();      hash = hash \* 31 + this.age;      return hash;    }  }  //FilteringSlicingStreams.java  import java.util.List;  import java.util.Arrays;  import static java.util.stream.Collectors.toList;  public class FilteringSlicingStreams {   static List<Employee> employeeList=Arrays.asList(     new Employee("Tom Jones", 45),     new Employee("Harry Major", 25),     new Employee("Ethan Hardy", 65),     new Employee("Nancy Smith", 15),     new Employee("Deborah Sprightly", 29),     new Employee("Dick Hiddleton Sprightly", 44),     new Employee("Alexander Hick", 19),     new Employee("Harry Major", 25),     new Employee("Nancy Smith", 15));   public static void main(String args[]){     //filter method     List<Employee> filteredList= employeeList.stream().filter((Employee e)-> e.getAge() > 24).collect(toList());     System.out.println("After applying filter method");     filteredList.forEach(System.out::println);     //distinct method     filteredList= filteredList.stream().distinct().collect(toList());     System.out.println("After applying distinct() method");     filteredList.forEach(System.out::println);     //limit method     filteredList= filteredList.stream().limit(2).collect(toList());     System.out.println("After applying limit(2) method");     filteredList.forEach(System.out::println);     //skip method     filteredList= filteredList.stream().skip(1).collect(toList());     System.out.println("After applying skip(1) method");     filteredList.forEach(System.out::println);   }  } |

**OUTPUT of the above code**

After applying filter() method  
Employee Name:Tom Jones Age:45  
Employee Name:Harry Major Age:25  
Employee Name:Ethan Hardy Age:65  
Employee Name:Deborah Sprightly Age:29  
Employee Name:Dick Hiddleton Sprightly Age:44  
Employee Name:Harry Major Age:25  
  
After applying distinct() method  
Employee Name:Tom Jones Age:45  
Employee Name:Harry Major Age:25  
Employee Name:Ethan Hardy Age:65  
Employee Name:Deborah Sprightly Age:29  
Employee Name:Dick Hiddleton Sprightly Age:44  
  
After applying limit(2) method  
Employee Name:Tom Jones Age:45  
Employee Name:Harry Major Age:25  
  
After applying skip(1) method  
Employee Name:Harry Major Age:25

**Explanation of the java example’s code & output**

* 2 classes are defined in the code above. First is **Employee**class which consists of 2 attributes - **name** & **age**. It also contains **equals()**& **hashcode()** methods implemented.
* Second class is **FilteringSlicingStreams**which defines a list of 9 Employees using **Arrays.asList()**.
* In the **main()** method first the **filter()** method is called with **Predicate** lambda defined in such a way that only employees greater than 24 years of age satisfy the condition. As a result the **filteredList**variable gets a list of Employees with 6 employees in them. I have highlighted the different portions of output with green color for easy readability. There are 6 employee records printed in the output below the line "After applying filter method".
* Next the **filteredList** is taken and **distinct()** method is applied to it. This list of 6 filtered employees is now reduced to 5 with 1 instance of Harry Major aged 25 being removed as it is duplicate. Note that [**hashcode()** & **equals()**](https://www.javabrahman.com/corejava/understanding-equals-and-hashcode-contract-when-using-collections-in-java/) implementation in **Employee** has been done to make distinct method work correctly. The list of 5 Employees remaining as a result is printed in the output below the line "After applying distinct() method".
* Next the **limit(2)** method is applied to the **filteredList** instance returned after applying distinct. This limits the resultant list to exactly 2 **Employee** records which are shown in the output below the line "After applying limit(2) method".
* Lastly the **skip(1)** method is applied to the **filteredList** which now contains 2 employees. This causes the resultant stream and list to skip the first **Employee** in the list(Tom Jones) and we are now left with only 1 **Employee** in the list(Harry Major). Details of this 1 employee are printed below the output line "After applying skip(1) method".

**Summary**: This wraps-up the tutorial on filtering and slicing of Streams using the filter, distinct, limit & skip methods.

**Java 8 Mapping with Streams | map and flatMap methods tutorial with examples**

[December 20, 2015](https://www.javabrahman.com/java-8/java-8-mapping-with-streams-map-flatmap-methods-tutorial-with-examples/) [0 Comments](https://www.javabrahman.com/java-8/java-8-mapping-with-streams-map-flatmap-methods-tutorial-with-examples/#disqus_thread)

**Introduction**:Java 8 Mapping with Streams tutorial explains the concept of mapping with streams using the map & flatMap methods with examples to show their usage. It assumes that you are familiar with basics of [Java 8 Streams API](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/).

**What is mapping with Streams**  
Mapping in the context of Java 8 Streams refers to converting or transforming a Stream carrying one type of data to another type.

Lets take an example. Say we have a stream containing elements of type **String**. Suppose what we need is a stream of data which contains **Integer** values with each **Integer** value of the new stream being the length of corresponding **String** in the old stream. I.e we need to convert **Stream<String>** to its corresponding**Stream<Integer>**.

Such conversion from one type to another is possible with the **map()** method of streams. On top of the **map()**method, streams provide a **flatmap()** method which helps in flattening *and* converting multi-level streams into a single stream. Lets see both of these methods in detail now.

**Definition & usage of map() method**  
The **map()** method helps us transform one type of stream to another type of stream.  
**Definition of map() method** in **java.util.stream.Stream<T>** is –

**<R> Stream<R> map(Function<? super T,? extends R> mapper)**

Where,

* **map()** method takes as input an instance of [Function <T, R>](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/) which converts the type of elements in the stream from the current type T to the new type R.
* It returns as output a stream of type **R** or **Stream<R>**

**Usage of map() method** is shown below with an example –

**Java 8 code showing Stream.map() method usage**

**public class Employee{**

**private String name;**

**private Integer age;**

**public Employee(String name, Integer age){**

**this.name=name;**

**this.age=age;**

**}**

**//Standard setters, getters, equals & hashcode**

**}**

**import static java.util.stream.Collectors.toList;**

**import java.util.Arrays;**

**import java.util.List;**

**public class StreamsMapping {**

**static List<Employee> employeeList =**

**Arrays.asList(new Employee("Tom Jones", 45),**

**new Employee("Harry Major", 25),**

**new Employee("Ethan Hardy", 65),**

**new Employee("Nancy Smith", 15),**

**new Employee("Deborah Sprightly", 29));**

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

**List<String> mappedList = employeeList.stream().**

**map(emp -> emp.getName()).**

**collect(toList());**

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

**}**

**}**

**OUTPUT of the main() method**

Tom Jones  
Harry Major  
Ethan Hardy  
Nancy Smith  
Deborah Sprightly

**Salient points of above map() method example**

* **StreamsMapping** class run’s the example in its **main()** method & contains a static list of **Employee** type of objects.
* **map()** method is used to convert the Stream of Employees to a Stream of Strings with just the employee names.
* Conversion from **Employee** type to **String** type is achieved using a **Function** instance defined by the [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) – **emp -> emp.getName()**
* The output obtained on printing the **mappedList** contents are all the employee names.

**Definition & usage of flatMap() method**  
The **flatMap()** method gives us the ability to flatten a multi-level stream obtained as a result of mapping each element of the input stream to a stream, and then creating a single stream out of this stream of streams.  
**Definition of flatMap() method** in **java.util.stream.Stream<T>** is –

**<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)**

Where,

* **flatMap()** method takes as input an instance of **Function<T,R>**, named **mapper**, which converts the type of elements in the stream from the current type **T** to another stream of type **R**
* It returns as output a stream of type **R** which is actually a flattened stream containing the elements in all the streams of type **R** obtained when **mapper** is applied on the input stream’s elements of type **T**.
* For n elements in input stream of type **T** we first get n streams of type **R**. Then these n streams are flattened into a single stream of type **R**. So, a **Stream<Stream<R>>** becomes **Stream<R>**.

**Usage of flatMap() method** is shown below with an example –  
*Note*– the **StreamsMapping** class used in the previous example is same, as also the **Employee** list defined in it. For brevity, I have included just the **main()** method here –

**Java 8 code showing Stream.flatMap() method usage**

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

**List<String> nameCharList = employeeList.stream()**

**.map(emp-> emp.getName().split(""))**

**.flatMap(array->Arrays.stream(array))**

**.map(str -> str.toUpperCase())**

**.filter(str -> !(str.equals(" ")))**

**.collect(toList());**

**nameCharList.forEach(str -> System.out.print(str));**

**}**

**}**

**Output of the main() method is**

TOMJONESHARRYMAJORETHANHARDYNANCYSMITHDEBORAHSPRIGHTLY

**Salient points of above flatMap() method example**

* The objective of the above code is to get all the alphabets used in the names of all the employees in caps for making their embossed name-plates.
* **map(emp-> emp.getName().split(""))** converts the stream of Employees to a stream of Employee names’ array. I.e. converts from **Stream<Employee>** to **Stream<String[]>**.
* **flatMap()** method converts **Stream<String[]>** into first a **Stream<Stream<String>>** and then flattens it to **Stream<String>** i.e. a stream of single character Strings.
* **map()** is applied next, which converts this **Stream<String>**’s contents to uppercase and then filters out the space characters using the **filter()** method.
* We have the required list at the end which contains all the alphabets in caps for all the employees, which is the output obtained on printing the **namecharlist**.

*Note* – there are other variants of **map()** & **flatMap()** methods which work with **DoubleStreams**, **LongStreams** & **IntStreams**. These are more specific forms of the general **map()** & **flatMap()** I have explained above. These have not been covered in this tutorial.

**Summary**  
This tutorial explained what is meant by Mapping in Streams followed by definition & usage of **map()** & **flatMap()** methods with examples.

**Java 8 Matching with Streams | allMatch, anyMatch, noneMatch methods tutorial with examples**

[September 10, 2016](https://www.javabrahman.com/java-8/java-8-matching-with-streams-allmatch-anymatch-nonematch-methods-tutorial-examples/)

**Introduction** – Java 8 Matching with Streams tutorial explains how to match elements in a stream using the **allMatch()**, **anyMatch()** and **noneMatch()** methods provided by the Streams API with examples to show their usage. This tutorial assumes that you are familiar with basics of [Java 8 Streams API](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/).

**What is ‘matching’ in the context of Streams**  
Given a stream of objects, many-a-times we need to check whether object(s) in the given stream match a specific criteria. Instead of writing logic for iterating over the stream elements and checking each object whether it matches the criteria (which is more of an [imperative rather than functional](https://www.javabrahman.com/programming-principles/imperative-versus-functional-programming-paradigms-conceptual/) style of programming), Java 8 Streams allow declarative matching of objects in the stream.

I.e. once you define the condition using a Predicate instance, and provide this Predicate as an input to the matching methods, then Java 8 processes the matching function internally and provides you with the result whether a match for the condition was found or not.

Java 8 provides such declarative matching with predicate conditions using three methods defined on the Streams API which are – **allMatch()**, **anyMatch()** and **noneMatch()**.

**Stream.allMatch() method**  
**Stream.allMatch()** method returns **true**if all the elements of the stream match the provided predicate condition. If even one of the elements does not match the predicate condition then the method skips the testing of the remaining elements using the concept of [short-circuit evaluation](https://www.javabrahman.com/programming-principles/short-circuiting-or-short-circuits-in-boolean-evaluations-in-programming-and-java/) and returns **false**as the result. This is a [terminal](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/) stream operation.

**Definition of allMatch() Method** – The **Stream.allMatch()** method has the following signature –

**boolean allMatch(Predicate<? super T> predicate)**

Where,  
     – input is **predicate** which is an instance of a [Predicate Functional Interface](https://www.javabrahman.com/java-8/java-8-java-util-function-predicate-tutorial-with-examples/)  
     – **boolean** value is returned indicating whether all elements of the stream match the **predicate** or not.

The below code shows how to use the **allMatch()** method in your code.

**Java 8 code showing Stream.allMatch() method usage**

**package com.javabrahman.java8;**

**public class Employee{**

**private String name;**

**private Integer age;**

**public Employee(String name, Integer age){**

**this.name=name;**

**this.age=age;**

**}**

**//getters and setters for name and age attributes go here**

**//overridden equals() and hashcode() go here**

**public String toString(){**

**return "Employee Name:"+this.name**

**+" Age:"+this.age;**

**}**

**}**

**//MatchingWithStreams.java**

**public class MatchingWithStreams {**

**static List<Employee> employeeList = Arrays.asList(new Employee("Tom Jones", 45),**

**new Employee("Harry Major", 25),**

**new Employee("Ethan Hardy", 65),**

**new Employee("Nancy Smith", 22),**

**new Employee("Deborah Sprightly", 29));**

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

**boolean allEmpAbove21 = employeeList.stream()**

**.allMatch(emp -> emp.getAge() > 21);**

**System.out.println("All employees are above 21:" + allEmpAbove21);**

**}**

**}**

**OUTPUT of the above code**

All employees are above 21:true

**Explanation of the code**

* **Employee** is the class of which we will be creating a **Stream**. It has two main attributes – **name**and **age**.
* **employeeList**is a static list of 5 **Employee**s.
* In the **main()** method we create a **Stream**of **Employee**s using the **stream()** method of **List**interface.
* On the stream of Employees we call the **allMatch()** method with the **Predicate** instance being specified as its equivalent [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) – **emp -> emp.getAge() > 21**. This predicate condition states that the Employee’s age should be greater than 21 years.
* Since, all the employees are of age greater than 21 years, so the **Predicate**is satisfied for all the **Employee**s, and the **allMatch()** method returns **true**.

**Stream.anyMatch() method**  
**Stream.anyMatch()** method returns **true** if at least 1 of the elements of the stream match the provided predicate condition. If none of the elements match the predicate condition then the method returns **false**. The moment this method finds the first element satisfying the predicate, it skips the testing of the remaining elements using the concept of short-circuit evaluation and returns **true** as the result. This is a terminal stream operation.

**Definition of anyMatch() Method** – The **Stream.anyMatch()** method has the following signature –

**boolean anyMatch(Predicate<? super T> predicate)**

Where,  
     – input is **predicate** which is an instance of a Predicate Functional Interface  
     – **boolean** value is returned indicating whether any of the elements of the stream match the **predicate** or not.

The below code shows how to use the **anyMatch()** method in your code.  
*(Note – The****Employee****class and****employeeList****objects with their values remain the same as the previous code usage example and hence are not shown below for brevity.)*

**Java 8 code showing Stream.anyMatch() method usage**

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

**boolean empAbove40 = employeeList.stream()**

**.anyMatch(emp -> emp.getAge() > 40);**

**System.out.println("Any employee is above 40:" + empAbove40);**

**}**

**OUTPUT of the above code**

Any employee is above 40:true

**Explanation of the code**

* In the **main()** method we create a **Stream**of **Employee**s using the **stream()** method of **List**interface.
* On the stream of Employees we call the **anyMatch()** method with the **Predicate** instance being specified as its equivalent lambda expression – **emp -> emp.getAge() > 40**. This predicate condition states that the Employee’s age should be greater than 40 years.
* Since, there are 2 employees of age greater than 40 years, the predicate is satisfied and the **anyMatch()**method returns **true**.

**Stream.noneMatch() method**  
**Stream.noneMatch()** method returns **true** if none of the elements of the stream match the provided predicate condition. If one (or more) of the elements match the predicate condition then the method returns **false**. The moment this method finds the first element satisfying the predicate, it skips the testing of the remaining elements using the concept of short-circuit evaluation and returns **false** as the result. This is a terminal stream operation.

**Definition of noneMatch() Method** – The **Stream.noneMatch()** method has the following signature –

**boolean noneMatch(Predicate<? super T> predicate)**

Where,  
     – input is **predicate** which is an instance of a Predicate Functional Interface  
     – **boolean** value is returned indicating whether none of the elements of the stream match the **predicate**.

The below code shows how to use the **noneMatch()** method in your code.  
*(Note – The****Employee****class and****employeeList****objects with their values remain the same as the previous code usage example and hence are not shown below for brevity.)*

**Java 8 code showing Stream.noneMatch() method usage**

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

**boolean noEmpBelow30 = employeeList.stream()**

**.noneMatch(emp -> emp.getAge() < 30);**

**System.out.println("No employee is below 30:" + noEmpBelow30);**

**}**

**OUTPUT of the above code**

No employee is below 30:false

**Explanation of the code**

* In the **main()** method we create a **Stream** of **Employee**s using the **stream()** method of **List**interface.
* On the stream of Employees we call the **noneMatch()** method with the **Predicate** instance being specified as its equivalent lambda expression – **emp -> emp.getAge() < 30**. This predicate condition states that the Employee’s age should be less than 30 years.
* Since, there are 3 employees of age less than 30 years, the **noneMatch()** method returns **false**. For the method to return **true** – none of the employees in the **employeeList** should have age less than 30 years.

**Conclusion** – In this tutorial we learnt what is meant by matching in the context of Java 8 Streams. Next we looked at the definition and code usage of three methods provided by Streams API for the purpose of matching – **allMatch()**, **anyMatch()** and **noneMatch()**.

**Java 8 Streams API – findAny, findFirst methods tutorial with examples**

[September 16, 2016](https://www.javabrahman.com/java-8/java-8-finding-with-streams-findfirst-findany-methods-tutorial-examples/)

**Introduction** – This tutorial explains the Java 8 **Stream API’s** **findAny()** and **findFirst()** methods with examples to show their usage. The tutorial assumes that you are familiar with basics of [Java 8 Streams API](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/).

**Stream.findAny() method**  
There are instances when the business specification says that any element of the stream satisfying a given criteria is to be fetched. I.e. an exact element match is not needed but any element from the satisfying set can be picked up. In such cases **findAny()** method is an ideal fit because once you [filter the stream](https://www.javabrahman.com/java-8/java-8-filtering-and-slicing-streams-tutorial-with-examples/) down to elements satisfying a particular criteria, then any element from the filtered stream can be picked up.

It is important to note that **Stream.findAny()** method can literally give you any element from the stream on which it is called. I.e. you should not code with an expectation that a particular value will always be returned. This *non-deterministic* nature of the **findAny()** method is very useful when executing parallel operations on a stream as it helps in performance optimisation without worrying about which element will be returned.

Also, the **Stream.findAny()** method is a [short-circuiting](https://www.javabrahman.com/programming-principles/short-circuiting-or-short-circuits-in-boolean-evaluations-in-programming-and-java/) and [terminal](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/) operation.

**Definition of Stream.findAny() Method** – The **Stream.findAny()** method has the following signature –   **Optional<T> findAny()**  
Where,  
     – **findAny()** method does not take any input.  
     – returns an **Optional** instance of type T where T is the type of the Stream on which this method is invoked.

The below code shows how to use the **Stream.findAny()** method in your code.

**Java 8 code showing Stream.findFirst() method usage**

**package com.javabrahman.java8;**

**public class Employee{**

**private String name;**

**private Integer age;**

**public Employee(String name, Integer age){**

**this.name=name;**

**this.age=age;**

**}**

**//getters and setters for name and age attributes go here**

**//overridden equals() and hashcode() go here**

**public String toString(){**

**return "Employee Name:"+this.name**

**+" Age:"+this.age;**

**}**

**}**

**//FindInStreams.java**

**package com.javabrahman.java8.streams;**

**import com.javabrahman.java8.Employee;**

**import java.util.Arrays;**

**import java.util.List;**

**import java.util.Optional;**

**public class FindInStreams {**

**static List<Employee> employeeList = Arrays.asList(**

**new Employee("Tom Jones", 45),**

**new Employee("Harry Major", 25),**

**new Employee("Ethan Hardy", 65),**

**new Employee("Nancy Smith", 22),**

**new Employee("Deborah Sprightly", 29),**

**new Employee("Billy Kid", 22),**

**new Employee("George King",44),**

**new Employee("Annie Barrey", 19));**

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

**Optional<Employee> anyEmpAbove40 = employeeList.stream()**

**.filter(emp -> emp.getAge() > 40)**

**.findAny();**

**if(anyEmpAbove40.isPresent()){**

**System.out.println("Any Employee above age 40: " + anyEmpAbove40.get());**

**}**

**}**

**}**

**OUTPUT of the above code**

Any Employee above age 40:  Employee Name:Tom Jones   Age:45

**Explanation of the code**

* **Employee** class is defined with two attributes – **name** and **age**.
* The **FindInStreams** class creates a static list of 8 **Employee** objects named **employeeList**.
* In the **main()** method of the **FindInStreams** class the **employeeList** is converted to a stream and filtered such that it contains only employees with age > 40.
* On the filtered stream **findAny()** method is invoked which returns an instance of **Optional** class named – **anyEmpAbove40**.
* The **get()** method of **Optional** is invoked on to get the **Employee** object in **anyEmpAbove40** after checking for a value’s presence using **Optional**’s **isPresent()** method.
* As seen in the output – **Tom Jones** with age **45** is the employee fetched using the **findAny()** method.

**Stream.findFirst() method**  
There are cases where the business specification requires the first element of a filtered stream to be fetched. This method is useful when the stream being worked on has a defined encounter order. In such cases getting the first element becomes a simple method call using the **Stream.findFirst()** method. This method is a short-circuiting terminal operation.

**What is encounter order in Streams**

Encounter order simply refers to the order in which the elements of a stream are processed. A stream made from an ordered source such as a **List** will have the encounter order defined as per the ordering of the elements in the source **List**. On the other hand, a stream made from a source which does not have an intrinsic order of elements defined, such as a **HashSet**, will not have any specific encounter order defined. Also, even if the initial stream’s elements are unordered, an encounter order can be introduced using **Stream** methods such as **sorted()**.

**Definition of Stream.findFirst() Method** – The **Stream.findAny()** method has the following signature – **Optional<T> findFirst()**  
Where,  
     – **findFirst()** method does not take any input.  
     – returns an **Optional** instance of type T where T is the type of the Stream on which this method is invoked.

The below code shows how to use the **Stream.findFirst()** method in your code.  
*Note – Employee class and the static Employee list is the same as above example and hence has not been shown again for brevity.*

**Java 8 code showing Stream.findFirst() method usage**

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

**Optional<Employee> firstEmpBelow30 = employeeList.stream()**

**.filter(emp -> emp.getAge() < 30)**

**.findFirst();**

**if (firstEmpBelow30.isPresent()) {**

**System.out.println("First employee below 30: " + firstEmpBelow30.get());**

**}**

**}**

**OUTPUT of the above code**

First employee below 30:  Employee Name:Harry Major   Age:25

**Explanation of the code**

* The **Stream** of **Employee** objects is filtered such that it contains only employees of age < 30.
* The **findFirst()** method is invoked which returns the first element of the filtered stream as per the encounter order, or original list order, as an instance of **Optional** class named – **firstEmpBelow30**.
* The **get()** method of **Optional** is invoked on **firstEmpBelow30** to get the **Employee** object in it after checking for a value’s presence using **Optional**’s **isPresent()** method.
* As seen in the output – **Harry Major** with age **25** is the employee fetched using the **findFirst()** method.

**Summary**  
In this tutorial we understood the **Stream.findAny()** and **Stream.findFirst()** methods of the Java 8 Streams API and saw code examples of their usage. We also understood the concept of ***encounter order*** in Streams.

**Java 8 Streams API – creating infinite streams with iterate and generate methods**

[September 17, 2016](https://www.javabrahman.com/java-8/java-8-streams-api-creating-infinite-streams-with-iterate-and-generate-methods/)

**Introduction** – This tutorial explains how to create infinite streams using the Java 8 **Stream API’s** **iterate()**and **generate()** methods with examples to show their usage. This tutorial assumes that you are familiar with basics of [Java 8 Streams API](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/).

**Infinite Streams**  
Streams are different from collections although they can be created from collections. Unlike collections, a stream can go on generating/producing values forever. Java 8 Streams API provides two static methods in the Stream interface for creating infinite streams. These are **Stream.iterate()** and **Stream.generate()**.

Since infinite streams need to be limited to a finite number, based on specific requirement, hence it is a common practice to limit the number of elements produced by a stream using the **Stream.limit()** method.

Let us now take a look at how **Stream.iterate()** and **Stream.generate()** methods can be used to produce infinite streams.

**Creating infinite Streams using the Stream.iterate() method**  
Let us start by looking at the signature of **Stream.iterate()** method –

**static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)**

Where,  
     – first input parameter is a seed value or initial value of type T  
     – second input parameter is a **UnaryOperator** function of type T  
     – output is a **Stream** of type T

**Stream.iterate()** method works just like a ***function-of*** algebraic operation which is commonly written as **ƒ(x)**. The method first returns the seed-value itself. For the 2nd element in the Stream it finds **ƒ(seed-value)** and from then on iteratively keeps applying ***function-of*** to the returned values.  
So,  
The **1st value** in the infinite **Stream<T>** will be the **seed-value**  
The **2nd value** will be **ƒ(seed-value)**.  
The **3rd value** will be **ƒ(ƒ(seed-value))**  
The **4th value** will be **ƒ(ƒ(ƒ(seed-value)))** ***and so on***…

Let us take an example to understand how **Stream.iterate()** method works –  
Suppose the **UnaryOperator<T>** function **fsqr()** is a square function defined using the lambda expression – **(Integer n) -> n\*n** and the **seed** is **2**.  
So,  
The **1st value** returned in the infinite stream will be **2**  
The **2nd value** returned in the stream will be **fsqr(2) OR 2\*2=4**.  
The **3rd value** will be fsqr(fsqr(2)) i.e. **fsqr(4) OR 4\*4=16**  
The **4th value** will be fsqr(fsqr(fsqr(2))) i.e. **fsqr(16) OR 16\*16=256** ***and so on***…

The infinite stream will then have values – **[2,4,16,256,… and so on…]**

The Java code for using **Stream.iterate()** method to produce a Stream of iteratively squared values will be as below –

**Java 8 code to produce an infinite Stream using Stream.iterate()**

**package com.javabrahman.java8.streams;**

**import java.util.stream.Stream;**

**public class InfiniteStreams {**

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

**Stream.iterate(2, (Integer n) -> n\*n)**

**.limit(5)**

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

**}**

**}**

**OUTPUT of the above code**

2  
4  
16  
256  
65536

**Explanation of the code**

* **Seed**passed as input to the **Stream.iterate()** method is **2**.
* **UnaryOperator** function instance is passed using the lambda expression- **(Integer n) -> n\*n**.
* The output stream is limited to **5** elements using **Stream.limit()** method.
* Output **Stream**is as expected with values – **2,4,16,256,65536**.

**Creating infinite Streams using the Stream.generate() method**  
**Stream.generate()** method generates an infinite stream of elements by repeatedly invoking a [Supplier Functional Interface](https://www.javabrahman.com/java-8/java-8-java-util-function-supplier-tutorial-with-examples/) instance passed to it as an input parameter. **Stream.generate()** method’s signature looks like this –

**static<T> Stream<T> generate(Supplier<T> s)**

Where,  
     – Only input is an instance of a **Supplier Functional Interface** of Type T  
     – Output is a **Stream** of type T

Let us now look at how to write the code to create an infinite stream containing random values using **Stream.generate()** and **Math.random()** methods.

**Java 8 code to produce an infinite Stream using Stream.generate()**

**package com.javabrahman.java8.streams;**

**import java.util.stream.Stream;**

**public class InfiniteStreams {**

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

**Stream.generate(Math::random)**

**.limit(5)**

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

**}**

**}**

**OUTPUT of the above code**

0.8756068395647292  
0.7717064739685572  
0.8199061254640724  
0.6481411588818413  
0.8075238156216996

**Explanation of the code**

* **Math.random()** method generates a random value between **0.0** and **1.0** every time it is called. The [function descriptor](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/) of **Math.random()** method matches that of the Supplier Functional Interface, so **Math.random()** is passed as an input to **Stream.generate()** method using its [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/) – **Math::random**.
* **Stream.generate()** method invokes **Math.random()** repeatedly to produce an infinite **Stream** of values between **0.0** and **1.0**.
* The output stream is limited to **5** elements using **Stream.limit()** method.
* Output **Stream**is as expected with 5 random values as shown above..

**Summary**  
In this tutorial we looked at what are infinite streams and how they can be generated using the static **Stream.iterate()** and **Stream.generate()** methods with Java 8 code examples to understand their usage.

**Java 8 Streams API – creating infinite streams with iterate and generate methods**

[September 17, 2016](https://www.javabrahman.com/java-8/java-8-streams-api-creating-infinite-streams-with-iterate-and-generate-methods/)

**Introduction** – This tutorial explains how to create infinite streams using the Java 8 **Stream API’s** **iterate()**and **generate()** methods with examples to show their usage. This tutorial assumes that you are familiar with basics of [Java 8 Streams API](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/).

**Infinite Streams**  
Streams are different from collections although they can be created from collections. Unlike collections, a stream can go on generating/producing values forever. Java 8 Streams API provides two static methods in the Stream interface for creating infinite streams. These are **Stream.iterate()** and **Stream.generate()**.

Since infinite streams need to be limited to a finite number, based on specific requirement, hence it is a common practice to limit the number of elements produced by a stream using the **Stream.limit()** method.

Let us now take a look at how **Stream.iterate()** and **Stream.generate()** methods can be used to produce infinite streams.

**Creating infinite Streams using the Stream.iterate() method**  
Let us start by looking at the signature of **Stream.iterate()** method –

**static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)**

Where,  
     – first input parameter is a seed value or initial value of type T  
     – second input parameter is a **UnaryOperator** function of type T  
     – output is a **Stream** of type T

**Stream.iterate()** method works just like a ***function-of*** algebraic operation which is commonly written as **ƒ(x)**. The method first returns the seed-value itself. For the 2nd element in the Stream it finds **ƒ(seed-value)** and from then on iteratively keeps applying ***function-of*** to the returned values.  
So,  
The **1st value** in the infinite **Stream<T>** will be the **seed-value**  
The **2nd value** will be **ƒ(seed-value)**.  
The **3rd value** will be **ƒ(ƒ(seed-value))**  
The **4th value** will be **ƒ(ƒ(ƒ(seed-value)))** ***and so on***…

Let us take an example to understand how **Stream.iterate()** method works –  
Suppose the **UnaryOperator<T>** function **fsqr()** is a square function defined using the lambda expression – **(Integer n) -> n\*n** and the **seed** is **2**.  
So,  
The **1st value** returned in the infinite stream will be **2**  
The **2nd value** returned in the stream will be **fsqr(2) OR 2\*2=4**.  
The **3rd value** will be fsqr(fsqr(2)) i.e. **fsqr(4) OR 4\*4=16**  
The **4th value** will be fsqr(fsqr(fsqr(2))) i.e. **fsqr(16) OR 16\*16=256** ***and so on***…

The infinite stream will then have values – **[2,4,16,256,… and so on…]**

The Java code for using **Stream.iterate()** method to produce a Stream of iteratively squared values will be as below –

**Java 8 code to produce an infinite Stream using Stream.iterate()**

**package com.javabrahman.java8.streams;**

**import java.util.stream.Stream;**

**public class InfiniteStreams {**

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

**Stream.iterate(2, (Integer n) -> n\*n)**

**.limit(5)**

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

**}**

**}**

**OUTPUT of the above code**

2  
4  
16  
256  
65536

**Explanation of the code**

* **Seed**passed as input to the **Stream.iterate()** method is **2**.
* **UnaryOperator** function instance is passed using the lambda expression- **(Integer n) -> n\*n**.
* The output stream is limited to **5** elements using **Stream.limit()** method.
* Output **Stream**is as expected with values – **2,4,16,256,65536**.

**Creating infinite Streams using the Stream.generate() method**  
**Stream.generate()** method generates an infinite stream of elements by repeatedly invoking a [Supplier Functional Interface](https://www.javabrahman.com/java-8/java-8-java-util-function-supplier-tutorial-with-examples/) instance passed to it as an input parameter. **Stream.generate()** method’s signature looks like this –

**static<T> Stream<T> generate(Supplier<T> s)**

Where,  
     – Only input is an instance of a **Supplier Functional Interface** of Type T  
     – Output is a **Stream** of type T

Let us now look at how to write the code to create an infinite stream containing random values using **Stream.generate()** and **Math.random()** methods.

**Java 8 code to produce an infinite Stream using Stream.generate()**

**package com.javabrahman.java8.streams;**

**import java.util.stream.Stream;**

**public class InfiniteStreams {**

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

**Stream.generate(Math::random)**

**.limit(5)**

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

**}**

**}**

**OUTPUT of the above code**

0.8756068395647292  
0.7717064739685572  
0.8199061254640724  
0.6481411588818413  
0.8075238156216996

**Explanation of the code**

* **Math.random()** method generates a random value between **0.0** and **1.0** every time it is called. The [function descriptor](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/) of **Math.random()** method matches that of the Supplier Functional Interface, so **Math.random()** is passed as an input to **Stream.generate()** method using its [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/) – **Math::random**.
* **Stream.generate()** method invokes **Math.random()** repeatedly to produce an infinite **Stream** of values between **0.0** and **1.0**.
* The output stream is limited to **5** elements using **Stream.limit()** method.
* Output **Stream**is as expected with 5 random values as shown above..

**Summary**  
In this tutorial we looked at what are infinite streams and how they can be generated using the static **Stream.iterate()** and **Stream.generate()** methods with Java 8 code examples to understand their usage.

**Java 8 Reducing with Streams | reduce method tutorial with examples**

[September 21, 2016](https://www.javabrahman.com/java-8/java-8-reducing-with-streams-reduce-method-tutorial-with-examples/)

**Introduction** – Java 8 Reducing with Streams tutorial starts with explaining the **concept of reducing in Streams**. It then looks at the **Streams API’s** **reduce()** **method** and how it can be used to **perform reduction operations on streams of data**. **Java 8 code examples** are used to demonstrate the method’s usage. This tutorial assumes that the reader is familiar with basics of [Java 8 Streams API](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/).

**What is ‘reducing’ in the context of Streams**  
Reducing in the context of Java 8 Streams refers to the process of combining all elements in the stream repeatedly to produce a single value which is returned as the result of the reduction operation.

Given a stream of elements there could be various ways in which one can reduce (or combine) them to a single resultant value such as summation of all elements (for numeric types), finding the maximum element from among all the elements (based on the elements’ comparison order), and similar operations for combining multiple elements into a single resultant value.

**The primary requirement of any reduction operation’s logic** is that it should use two operands for the operation which are –

1. The collective value aggregated or derived from the elements encountered so far which will be of the same type as the type of elements in the stream.
2. The value which is encountered next as the unprocessed value in the stream.

Due to this inherent nature of reduction operations requiring two operands both of which are of the same type as the type of elements in the stream being processed, Stream API’s **reduce()** method also uses a BinaryOperator function for defining the reduction operation logic.

Let us now take a look at how the Stream API’s **reduce()** operation is defined and used.

**Stream.reduce() method**  
**Stream.reduce()** method has the following signature –

**T reduce(T identity, BinaryOperator<T> accumulator);**

Where,  
     – **identity** is initial value of type **T** which will be used as the first value in the reduction operation.  
     – **accumulator** is an instance of a **BinaryOperator** Function(Functional Interface) of type **T**.

How the **stream.reduce()** operation works is that it applies the **BinaryOperator** function’s logic to the elements of the stream repeatedly. **BinaryOperator** **operates on 2 operands of type** **T** where –

1. **The first operand** contains the reduced (or combined) value till the current processed value in the stream. When the reducing operation starts, the identity value supplied to the reduce() method is used as this operand’s value.
2. **The second operand** is the next unprocessed value obtained from the Stream.

*(Note that 1 & 2 exactly match reduction operation’s operand definition described above)*

**Stream.reduce()** is a [terminal operation](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/).

Let us now move to the **1st** example where the reduction operation sums up all elements of the stream. After the code I will explain in detail how the summation logic works using the **Stream.reduce()** method.

**Example 1: Finding aggregate of stream elements using Stream.reduce() method**

**Example 1-Java 8 code showing Stream.reduce() method for aggregation**

|  |
| --- |
| //Employee.java  package com.javabrahman.java8;  public class Employee{    private String name;    private Integer age;    private Double salary;    public Employee(String name, Integer age, Double salary){      this.name=name;      this.age=age;      this.salary=salary    }    //getters and setters for name and age attributes go here    //overridden equals() and hashcode() go here    public String toString(){      return "Employee Name: "+this.name        +"  Age: "+this.age        +"  Salary: "+this.salary;    }  }  //ReducingWithStreams.java  package com.javabrahman.java8.streams;  import com.javabrahman.java8.Employee;  import java.util.Arrays;  import java.util.List;  public class ReducingWithStreams {    static List<Employee> employeeList = Arrays.asList(        new Employee("Tom Jones", 45, 7000.00),        new Employee("Harry Major", 25, 10000.00),        new Employee("Ethan Hardy", 65, 8000.00),        new Employee("Nancy Smith", 22, 12000.00),        new Employee("Deborah Sprightly", 29, 9000.00));      public static void main(String[] args) {      Double totalSalaryExpense = employeeList.stream()                                 .map(emp -> emp.getSalary())                                 .reduce(0.00,(a,b) -> a+b);      System.out.println("Total salary expense: "+totalSalaryExpense);     }  } |

**OUTPUT of the above code**

Total salary expense:  46000.0

**Explanation of the code**

* **Employee** is the class of which we will be creating a **Stream**. It has two main attributes – **name** and **age**.
* **employeeList** is a static list of 5 **Employee**s.
* Our aim is to calculate the total salary expenses of the company for which we will find the sum of all employee’s salaries using **Stream.reduce()** method.
* We start off with creating a **Stream** of **Employee**s from **employeeList** using the **stream()** method of **List** interface.
* We then [map the Stream](https://www.javabrahman.com/java-8/java-8-mapping-with-streams-map-flatmap-methods-tutorial-with-examples/) of Employee objects into a **Stream** of *salaries of employees* of type **Double**.
* Next we use the **reduce()** method, pipelined to the **Stream** of **Double**-valued salaries, to start aggregating the salaries. We pass the [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) – **(a,b) -> a+b** for specifying the **BinaryOperator**function’s logic.
* This is how the reduction operation (of aggregation) will work as the **Double**-valued salaries are encountered by the **reduce()** method –
  + **1stcall** to the **BinaryOperator** function – **Sum of 0.00**(initial value) **+ 7000.00**(1st value from stream)
  + **2ndcall – 7000.00**(sum till now) **+ 10000.00**(2nd value from stream)
  + **3rdcall – 17000.00**(sum till now) **+ 8000.00**(3rd value from stream)
  + **4thcall – 25000.00**(sum till now) **+ 12000.00**(4th value from stream)
  + **5thcall – 37000.00**(sum till now) **+ 9000.00**(5th value from stream)
* The total salary expense of **46000.00** is returned as the resultant value of **stream()** operation and assigned to the **totalSalaryExpense** variable which is then printed as output.

Let us now take a look at the **2nd example** showing the **reduce()** **method usage**.This time we will find the employee with the maximum salary from among all employees.  
*(Note – The****Employee****class and static****employeeList****are same as the above example and hence have been left out of the code below for brevity)*

**Example 2: Using Stream.reduce() method for finding employee with maximum salary**

**Example 2-Stream.reduce()method for finding employee with maximum salary**

|  |
| --- |
| public static void main(String[] args) {  Optional<Employee> maxSalaryEmp=employeeList.stream()      .reduce((Employee a, Employee b) -> a.getSalary() < b.getSalary() ? b:a);  if(maxSalaryEmp.isPresent())    System.out.println("Employee with max salary: "+maxSalaryEmp.get());  } |

**OUTPUT of the above code**

Employee with max salary:    Employee Name: Nancy Smith   Age:22   Salary: 12000.0

**Explanation of the code**

* Our aim is to find the Employee object with the maximum value stored in the salary attribute using **Stream.reduce()** method.
* We start off with creating a **Stream** of **Employee**s from **employeeList** using the **stream()** method of **List** interface.
* Next we use the **reduce()** method, pipelined to the **Stream** of **Employee**objects, to start with the process of finding the employee with the maximum salary. We pass the lambda expression- **(Employee a, Employee b) -> a.getSalary()< b.getSalary() ? b : a** for specifying the **BinaryOperator**function’s logic.
* This example uses the overloaded variant of **reduce()** method which doesn’t take the initial value and returns an **Optional<Employee>** type of value containing the result.
* This is how the reduction operation (of finding maximum salary) will work as the salaries are encountered by the **reduce()** method –
  + **1stcall** to the **BinaryOperator** function – **returns maximum of 7000.00**(1st value from stream) **and 10000.00**(2nd value from stream)
  + **2ndcall – returns max of 10000.00**(max value till now) **and 8000**(3rd value from stream)
  + **3rdcall – returns max of 10000.00**(max value till now) **and 12000.00**(4th value from stream)
  + **4thcall – returns max of 12000.00**(max value till now) **and 9000.00**(5th value from stream)
* **12000.00** is the maximum value and the **Employee** object with this salary is returned inside an **Optional<Employee>** instance and assigned to **maxSalaryEmp** variable.
* We check for presence of value inside **maxSalaryEmp** using **Optional**’s **isPresent()** method and then print the **Employee** object obtained using **Optional.get()** method.

**Conclusion** – In this tutorial we first learnt what is meant by reducing in the context of Java 8 Streams. Next we looked at the **reduce()** method provided by Streams API for the purpose of performing reduction operations. Lastly, we understood the working of the **reduce()** method by looking at code examples for two different stream reduction operations of aggregation and finding the maximum value.

**Default Methods in Java 8 with examples**

[November 1, 2015](https://www.javabrahman.com/java-8/default-methods-in-java-8-with-examples/)

This tutorial explains what are default methods in Java 8 with examples, why they are useful and how they can be used to enhance the design of your code.

**A little background**: Prior to Java 8 interfaces could not have any implemented code. Methods defined in an interface for a *type* were considered to be the API methods for that *type* and any classes implementing that API had to implement all the methods. Well, the API part is still true in Java 8 i.e. interfaces will act as APIs for your design. However, interfaces *can* have implemented code in Java 8. These can be written in 2 ways – static methods & default methods. I have dealt with static methods in a [separate article](https://www.javabrahman.com/java-8/java-8-static-methods-vs-default-methods-in-interfaces/). This article will focus on default methods.

**What are Default Methods**  
Default methods are methods implemented in an interface, are non-abstract, and marked by the modifier **default**. These methods are available to all classes which implement this interface.

**Format of default methods**  
Inside an interface we need to mark a default method with the modifier **default** as shown in example below –

**Interface with a default method**

**//MyInterface.java**

**public interface MyInterface{**

**default void printHello(){**

**System.out.println("Hello Default");**

**}**

**}**

**//MyInterfaceImpl.java**

**public class MyInterfaceImpl implements MyInterface{**

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

**new MyInterfaceImpl().printHello();**

**}**

**}**

**OUTPUT obtained by running MyInterfaceImpl**

 Hello Default

**Few points to note about the above program**

* **MyInterface**implements a default method **printHello()**.
* Method **printHello()** is implemented to simply print **"Hello Default"**.
* **MyInterfaceImpl**class implements **MyInterface**
* In the main method of **MyInterfaceImpl** when **printHello()** method is invoked on an instance of **MyInterfaceImpl** then it calls the default implementation of **printHello()** and prints **"Hello Default"**

**How are default methods Useful**  
Default methods are useful in a big way for API designers and indirectly for API implementors. Lets see how. Lets say we define an interface for a group of handheld devices called **DeviceAPI**. It has methods like **getOS()**, **getMaxResolution()** and **getBrand()** – all abstract. Our interface **DeviceAPI** would then look like this –

**DeviceAPI.java**

**public interface DeviceAPI{**

**public String getOS();**

**public String getMaxResolution();**

**public String getBrand();**

**}**

Now, over a period of time many devices are rolled out and for each new device we implement the **DeviceAPI**interface and hence, we also implement all the abstract methods. Its all going fine when we come to know that a wearable device has come out which needs to have a check for whether bluetooth is enabled on the device.

We promptly write a method **isBlueToothEnabled()** in the interface **DeviceAPI**. But then we notice that **DeviceAPI** has had more than a few, infact 20-30 implementations, many of these implementations were done by device vendors using this API and we have no access to their source code. If we go ahead and add this new method **isBlueToothEnabled()** to **DeviceAPI**, everything will be fine for the vendors till they do not recompile their implementation class along with the new **DeviceAPI**. Then their code will start failing as compiler will not find the new method in their implementation. Secondly, we will have to change *all* of the 20-30 implementations. Even though that device maybe a wearable type of device or not.

**How to overcome the issue of adding a new functionality to the API without breaking the code**  
Till Java 7 this kind of new functionality was very difficult to introduce due to the problems we noted above. But in Java 8 we have a solution for this. **We can implement this isBlueToothEnabled() method as a default method in DeviceAPI.** This implementation will then automatically be available to all the classes which implement **DeviceAPI**.

The code for **DeviceAPI** with the default method would look like this –

**DeviceAPI.java with default method**

**public interface DeviceAPI{**

**default boolean isBlueToothEnabled(){**

**//default method implementation goes here**

**}**

**public String getOS();**

**public String getMaxResolution();**

**public String getBrand();**

**}**

**Default methods as optional**  
To continue with the previous example, our design was good and default methods solved the problem. However, many device vendors complained that their classes were unnecessarily getting this new method **isBlueToothEnabled()**. Vendors complain to us that their maintenance engineers may end up using this method as they might miss reading the documentation around this method. Basically, vendors want this method to be available if they need it, else they do not want to support this method for their devices.

As owners of **DeviceAPI** we then tweak the default implementation of **isBlueToothEnabled()** method to the one shown below –

**DeviceAPI.java with default method made optional to inherit**

**public interface DeviceAPI{**

**default boolean isBlueToothEnabled(){**

**throw new UnsupportedOperationException();**

**}**

**public String getOS();**

**public String getMaxResolution();**

**public String getBrand();**

**}**

**Lets understand how the above code solves our problem**

* **isBlueToothEnabled()** is the default method. However, it does not contain any implementation.
* **isBlueToothEnabled()** by default throws **UnsupportedOperationException**.
* Any class which implements **DeviceAPI** will not be forced with a default implementation if they do not need it. By default, this new operation **isBlueToothEnabled()** will be unsupported.
* Only when an implementation class really needs the **isBlueToothEnabled()** method they will override this method and write their own version
* A default method written in the above way thus becomes *optional*.

**Default Methods and multiple inheritance**  
A class can implement multiple interfaces and each of these interfaces can have default methods. Then this becomes an instance of multiple inheritance which was till Java 7 not possible in Java while it was possible in languages such as C++. Multiple inheritance in Java 8 using default methods is a topic on its own and I have written a separate tutorial on it here – [multiple inheritance tutorial](https://www.javabrahman.com/java-8/java-8-multiple-inheritance-of-behavior-from-interfaces-using-default-methods/).

**Summary**  
This concludes the tutorial on default methods where we understood what are default methods, how we implement default methods in Java 8 interfaces, what kind of a basic design extensibility default methods solve,had a look at making default methods optional and lastly saw that default methods enable multiple inheritance in Java.

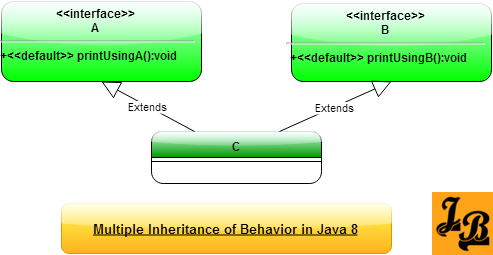
**Java 8 Multiple Inheritance of Behavior from Interfaces using Default Methods**

[November 7, 2015](https://www.javabrahman.com/java-8/java-8-multiple-inheritance-of-behavior-from-interfaces-using-default-methods/)

This article explains how default methods enable multiple inheritance of behavior in Java 8 using the new default methods feature in Interfaces.

**Multiple Inheritance of Behavior with Default Methods**  
Until Java 7 multiple interface inheritance was possible but interfaces were not allowed to have concrete methods. Thus, behavior inheritance was not possible. With Java 8, where interfaces can now have [default methods](https://www.javabrahman.com/java-8/default-methods-in-java-8-with-examples/) implemented in them, it is now possible to have a derived class inherit methods from multiple parent interfaces. So, multiple inheritance of behavior is now possible.

**Example to understand multiple inheritance using default methods**



In the above class diagram –

* A is an interface with a default implemented method **printUsingA()**.
* B is an interface with a default implemented method **printUsingB()**.
* C is a concrete class implementing both A and B interfaces.

Now, since C implements both A & B interfaces, it inherits the default methods of both the interfaces. An instance of C can thus be used to invoke default methods of A and B. Lets see code of such an invocation on class C below, starting off with code defining interfaces A & B –

**Java 8 code showing multiple inheritance for interfaces A,B & class C**

|  |
| --- |
| //A.java  public interface A{    public default void printUsingA(){      System.out.println("Print from A");    }  }  //B.java  public interface B{    public default void printUsingB(){      System.out.println("Print from B");    }  }  //C.java  public class C implements A,B{    public static void main(String args[]){      C cObj=new C();      cObj.printUsingA();      cObj.printUsingB();    }  } |

**OUTPUT of the above code**

Print from A  
Print from B

**Explanation of the code**

* In the **main()** method of C an instance of C called cObj is created.
* Methods **printUsingA()** & **printUsingB()** are invoked on cObj.
* Multiple inheritance of behavior is thus achieved as C inherits 2 implemented methods(or behaviors from A & B).

**Conflict Resolution in case of inheriting methods with same signature**  
We just saw above how C inherits implemented default methods from interfaces A & B. The two inherited methods had different names so there was no confusion. But what if the default methods being inherited had the same signatures – same name and same return types. Java 8 provides resolution rules for this.

**I have covered conflict resolution rules for such scenarios and the classic diamond problem resolution in my next article which you can**[read here](https://www.javabrahman.com/java-8/java-8-multiple-inheritance-conflict-resolution-rules-and-diamond-problem/)**.**

**Java 8 Multiple Inheritance Conflict Resolution Rules and Diamond Problem**

[November 8, 2015](https://www.javabrahman.com/java-8/java-8-multiple-inheritance-conflict-resolution-rules-and-diamond-problem/)

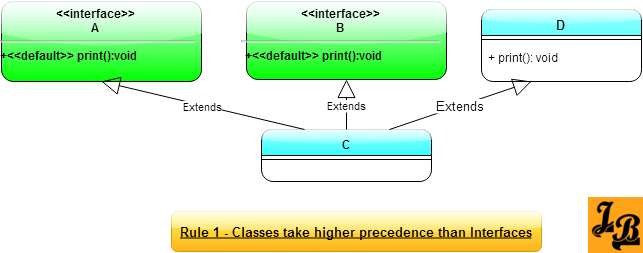
This article explains how to resolve conflicts when inheriting [default methods](https://www.javabrahman.com/java-8/default-methods-in-java-8-with-examples/) with same signatures from multiple interfaces using java 8’s conflict resolution rules. It then looks in to the classic Diamond Problem and its resolution in Java 8.

**What is default method resolution conflict**  
In my previous article I explained how Java 8 supports [multiple inheritance](https://www.javabrahman.com/java-8/java-8-multiple-inheritance-of-behavior-from-interfaces-using-default-methods/) of behavior using default methods. However, *what-if* the multiple default interfaces implemented have default methods with the same signatures. Then which of the default implementations from the many parent interfaces will be invoked in the implementing class.

Java 8 designers have thought of this conflict and have defined resolution rules for such scenarios. Let us now take a look at the possible conflict scenarios and the resolution rules in-built in Java 8 for avoiding them.

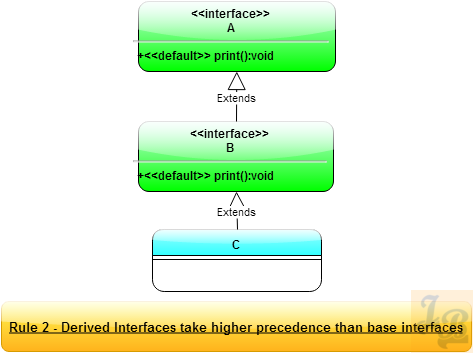
**Conflict resolution rules for inherited default methods**  
Conflict Resolution Rules for inherited default methods **in order of precedence are –**

* **Rule 1 – Classes take higher precedence than interfaces** – Any method inherited from a class or a superclass is invoked over any default method inherited from an interface.  
  *Example to explain Rule 1*



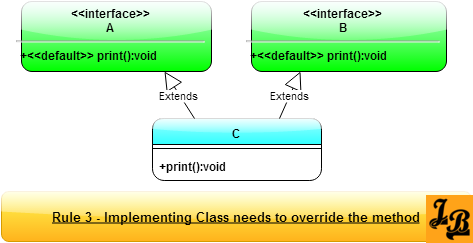
In the above class diagram, Class C inherits default method print() from interface A, interface B and super class C. If print() method is invoked in Class C then the implementation in super class C is executed.

* **Rule 2 – Derived interfaces or sub-interfaces take higher precedence than the interfaces higher-up in the inheritance hierarchy** – If default methods with the same method signature exist in an interface and its child interfaces, then the default method from the child interface is invoked.  
  *Example to explain Rule 2*



In the above class diagram, interface B inherits from interface A. Both have a default method print() with the same signature. Class C implements both interfaces A & B. When print() method is invoked on an instance of class C then the implementation in interface B is invoked as it is the lowest child/most derived interface in the inheritance hierarchy.

* **Rule 3 – In case Rule 1 and Rule 2 are not able to resolve the conflict then the implementing class has to specifically override and provide a method with the same method definition** – The implementing class can, of course, invoke the specific default method from the specific parent interface to get desired behavior. But still the class needs to override and invoke the default method to resolve the conflict.  
  *Example to explain Rule 3*



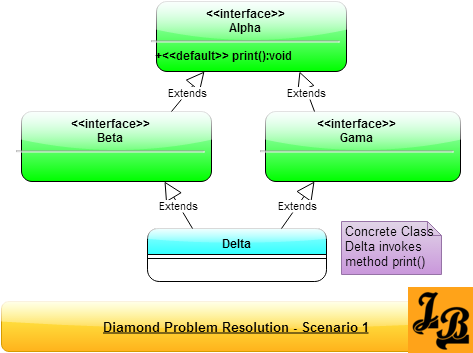
In the above class diagram, class C inherits from interfaces A & B, both of which have the default implementations of print(). Since, both interfaces A & B are parents of C, they are at the same hierarchy level, and hence, C has to provide its own implementation of method print().  
*Important Note* – Inside Class C’s implementation of print() method it should invoke the specific implementation of interface A or B. For this Java 8 has a special syntax –

**<super-interface-name>.super<method-name>**

In this case print() method in class C will invoke print() method of B,its parent, like this – **B.super.print()**

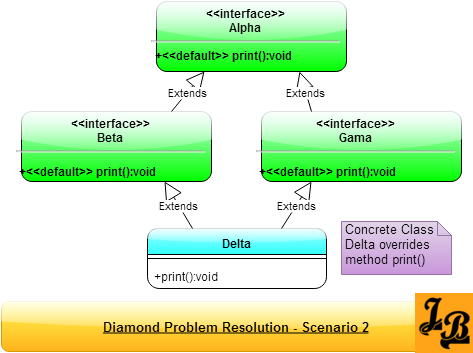
**Diamond Problem Revisited with Java 8’s Default Methods in Interfaces**  
Diamond Problem, wherein the grand-child class has both of its parents inheriting from a single grand-parent class, is a common problem faced in languages with multiple inheritance. Java 8 is affected by the diamond problem in 2 ways/2 scenarios which I will cover below –

* **Scenario 1 of diamond problem in Java 8** –



In the above class diagram, interfaces Beta & Gama implement interface Alpha. Class Delta is a concrete class and it implements both the interfaces Beta & Gama. The question in this scenario is that if Delta invokes the print() method, then which one will be invoked – the one inherited from Beta or that which was inherited from Gama?  
(*On a side note – If one sees the 4 inheritance arrows marked “extends” then they form a diamond shape and hence the name Diamond Problem*).  
**Resolution for Scenario 1 of Diamond Problem** –  
Java 8 resolves this situation by considering that there is only one implementation of print() method, which is in class Alpha. Hence, when Delta invokes print() then the implementation of print() in Alpha is executed.

* **Scenario 2 of diamond problem in Java 8** –



In the above class diagram, interfaces Beta & Gama implement interface Alpha like in scenario 1 above. However, here both of them override the print() method of Alpha. Now, when Delta wants to invoke print() which instance of print() will be invoked, the one inherited from Beta or the one inherited from Gama?  
**Resolution for Scenario 2 of Diamond Problem** –  
The answer lies in Rule 3 of the conflict resolution scenarios described earlier in this tutorial. Delta will have to override print() method and explicitly invoke the print() method on one of its parents – Beta or Gama. This will resolve the issue.

**Summary**  
In this tutorial we looked at what do we mean by conflict resolution when inheriting behavior from multiple interfaces using default methods and rules defined in Java 8 for resolving these conflicts. We then looked at couple of scenarios how Diamond Problem can occur in Java 8 along with how to resolve them in Java 8.

**Java 8 Static Methods vs Default Methods in Interfaces**

[November 3, 2015](https://www.javabrahman.com/java-8/java-8-static-methods-vs-default-methods-in-interfaces/)

This article explains Java 8 Static Methods and Default Methods in Interfaces w.r.t to their various aspects with examples. An important point to understand before we start looking at static and default methods in interfaces is that they are *not* an either/or options. Its not like you have to choose one of them to implement. On the contrary, static and default methods work together to deliver the full functionality of an interface.

**Quick Introductions to Static & Default Methods**  
**What are Static Methods**: These are methods written in Interfaces which are static. Till Java 7, static methods were only allowed on Classes. Beginning Java 8 static methods are now allowed in Interfaces as well.  
Static methods in interfaces are accessible through the interface name like this –  
**<Interface-Name>.<Static-Method-Name>**.  
For example:**java.util.Predicate** Interface has a static method **isEqual()** which can be accessed like this **Predicate.isEqual()**.  
Note – This is same as the way static methods are accessed on a Class.

**What are Default Methods**: These are methods written in Interfaces, indicated by the modifier **default**. As their name suggests these methods are available by default to all the classes which implement this interface. In case you are not familar with default methods, please refer the detailed article I have written on them here – [default methods tutorial](https://www.javabrahman.com/java-8/default-methods-in-java-8-with-examples/).

**Java 8 Static Methods vs Default Methods in Interfaces**  
Listed below are the main diftferences in the way static and default methods are used in interfaces –  
**Difference 1 – Definition in code\usage**  
Lets see how static and default methods are defined in code with an example –

**Java 8 code showing static and default methods' definition**

**//InterfaceWithDefaultStatic.java**

**public interface InterfaceWithDefaultStatic {**

**public static void staticMethod() {**

**System.out.println("Static Method's print");**

**}**

**public default void defaultMethod() {**

**System.out.println("Static Method's print");**

**}**

**}**

**A static method is defined similar to the way a static variable or method is defined in classes** –  
**<access modifier> static <return-parameter> method-name([<method-params>])[throws <Exceptions>]**  
Example: **public static void staticMethod()**

**A default method is defined in a very similar way except that instead of static keyword, the defaultkeyword is used** –  
**<access modifier> default <return-parameter> method-name([<method-params>])[throws <Exceptions>]**  
Example: **public default void defaultMethod()**

**Difference 2 – Scope for method invocation**  
**A static method is visible/usable in Interface Scope.** Once the interface has been compiled, then the static method can be invoked as –  
**<Interface-Name>.<static-method-name>([params])**  
For Example: For the code shown in difference-1 above, **staticMethod()** would be invoked as – **InterfaceWithDefaultStatic.staticMethod()**

**A default method is visible\usable in the object instance scope.** A Class needs to implement the interface containing the default method, then the default method can be invoked on the instance of the implementing class. The format of invocation of a default method is –  
**<ObjectInstance-name>.<default-method-name>([params])**  
For Example: Lets say there is a class named **DefaultImpl** as shown below –

**DefaultImpl.java**

**public class DefaultImpl implements InterfaceWithDefaultStatic{**

**//class contents**

**}**

For the above class **DefaultImpl**, to invoke the default method we will have to create an instance of the class, and then invoke it like this –

**Invocation of default method of DefaultImpl.java**

**DefaultImpl defaultImplInstance=new DefaultImpl();**

**defaultImplInstance.defaultMethod();**

**Difference 3 – Purpose served in the overall design**  
**When to design\use Static Methods**: Static Methods are the utility(util) methods which are associated to an Interface. So, in case you need any util methods which can operate on an Interface’s implementing class’s instances, then add that as a static method to the Interface itself.

Until Java 7, the general practice was to have such util methods in a separate utility class which contains all the static methods which can be invoked on an interface’s implementation’s instance.  
For example: Collections class has a sort method –  
**static <T> void sort(List<T> list, Comparator<? super T> c)**.  
This is how any list was sorted till Java 7. The sorting util methods is present in the Collections Class as a static method.

From Java 8 onwards, since Interfaces can have static methods, it makes the design more cohesive by keeping these static util methods inside List itself. The Java designers also thought of the same and added a static method **sort()** to List –  
**default void sort(Comparator<? super E> c)**  
**To summarize, whenever you need a static utility method for interface’s implementors, add that static utility method in Interface itself as a static method.**

**When to design\use Default Methods**: Default methods are used when a feature is added to an existing hierarchy of classes which is not needed for the whole hierarchy.In that case, making that method an abstract method makes it mandatory for all the classes in the hierarchy to implement. This is an unnecessary overhead and a maintenance nightmare.  
In this scenario, it is much simpler to add that method as a default method so that any Interface implementation which wants to use it can do so and the rest of the implementations can just ignore it. In effect making the functionality implemented through default methods an *optional* feature rather than a *mandatory* one.

**Summary**  
In the above article we looked at the basics of the new static and default methods in interfaces. Then we looked at the key differences in their usage from syntax, scope and design angles.

**Java 8 – java.util.stream.Collector basics tutorial with examples**

[February 13, 2017](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/)

Collectors play an important role in Java 8 streams processing. They ‘collect’ the processed elements of the stream into a final representation. Invoking the **collect()** method on a Stream, with a Collector instance passed as a parameter ends that Stream’s processing and returns back the final result.**Stream.collect()** is thus a [terminal operation](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/).

*But what exactly does a Collector do apart from ending a Stream and handing back the processed data? Which internal components of a Collector work in tandem to enable to produce the resulting Collection? Which collection operations are pre-defined as part of Java 8 language API?*

In this tutorial I will attempt to answer the above fundamental questions about Collectors. We will start with formally defining a Collector and its capabilities. Next, we will see various components of a Collector and understand how they work together. We will then look at the interface**java.util.stream.Collector** and understand how the components of Collector use the interface members. Next, we will have a quick overview of the commonly used predefined collectors defined in**java.util.Stream.Collectors** class. We will finish this tutorial with a Java code example showing a predefined Collector in action.

**What is java.util.stream.Collector – formal definition**  
The official java doc for **java.util.stream.Collector** defines it as[1](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/#ref1) –

*A Collector is a mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed.*

The above definition does seem a bit overbearing in terminology, at least for someone new to functional programming per say! Let me make an attempt to make it more comprehensible by stating it as below –

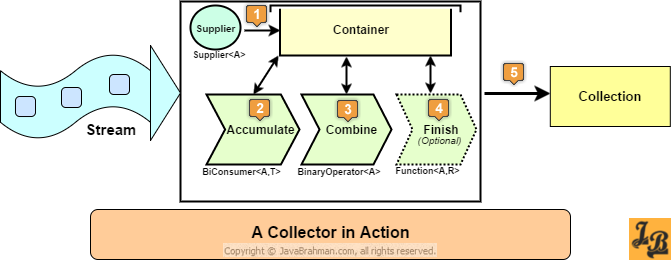
*A Collector is a terminal operation which reduces the stream being processed to a Collection. For this reduction it uses a single modifiable collection instance in which it adds all the processed stream elements as it encounters them. Collector also comes with the feature of optionally mapping the stream elements to an equivalent form using specified logic while they are being collected.*

I hope the above definition makes it clear what you can do with a Collector at least at a high level. We will of course take a deep dive into understanding Collectors, but before that I need to explain an important concept mentioned in the formal Collector definition above – that of a ***mutable reduction operation***.

**What is a mutable reduction operation**

A mutable reduction operation(such as **Stream.collect()**)collects the stream elements in a mutable result container(collection) as it processes them. Mutable reduction operations provide much improved performance when compared to an immutable reduction operation(such as **Stream.reduce()**). This is due to the fact that the collection holding the result at each step of reduction is mutable for a Collector and can be used again in the next step. **Stream.reduce()** operation, on the other hand, uses immutable result containers and as a result needs to instantiate a new instance of the container at every intermediate step of reduction which degrades performance.

**What (all) does a Collector do internally when it collects?**  
A collector collects the elements of a stream into a mutable container as we understood above. But internally it does a lot more than simply ‘collect’. Drawn next is a diagram showing a Collector in action as it collects –



As you can see in the above diagram there are broadly 5 steps*(marked by 5 orange markers)* which a typical Collector goes through while processing a Stream of elements. Let us quickly understand each of these steps –

* **Step 1 – *Supplier* provides the mutable empty result container**: Supplier is an instance of the [Supplier](https://www.javabrahman.com/java-8/java-8-java-util-function-supplier-tutorial-with-examples/)functional interface which provides an instance of a Collection(or Map) to hold the collected elements.
* **Step 2 – *Accumulator* adds individual elements into the result container**: Accumulator is an instance of **BiConsumer** functional interface. It adds individual elements of stream encountered by it into the result container. Accumulation action in this step is known as a fold in functional programming parlance.
* **Step 3 – *Combiner* combines two partial results**: Combiner is an instance of a **BinaryOperator** functional interface which combines two partial results returned by two separate groups of accumulations done in parallel.
* **Step 4 – Optional *Finisher* to put the processed elements in a desired form**: Finisher is an instance of a [Function](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/) interface and its use is optional. If required, a Finisher can be used to map the collected elements in the result container to a different required form.
* **Step 5 – Final Result**: The final collected elements are returned by the Collection in the result container i.e. Collection instance.

Having seen the four important components of a Collector, viz. Supplier, Accumulator, Combiner and Finisher, it is time now to get introduced to the interface – **Collector<T,A,R>**.

**Collector interface**  
(**java.util.stream.Collector**) interface is defined as follows –

**public Interface Collector<T,A,R>**

Where,  
     – **T** is element type being processed by the Stream and is to be ‘collected’  
     – **A** is the type of the accumulated result container which keeps on getting elements (of type **T**) added throughout the ‘collecting’ process.  
     – **R** is the type of the result container, or the collection, which is returned back as the ‘final’ output by the collector

**How Collector interface members are used by 4 components of a Collector**

* ***Supplier*** provides empty instance(or instances for parallel collectors) of type A to begin the accumulation of elements
* ***Accumulator*** uses an instance of A to collect T.
* ***Combiner*** combines two partial accumulated results of type A to produce a combined instance of A.
* ***Finisher*** maps A to R using a mapping function.

So far, we have understood the components of a Collector, and how these components work together and produced the final results. *At this point you may be wondering whether for simple tasks, such as collecting the processed elements of a Stream, you will need to implement so many types and components*? The good news is – *you need not!* This is where the predefined collectors come in handy.

**Predefined collectors overview**  
Java 8 designers have thought of the most common mutable reduction operations which might be required by application developers. Implementation for these operations have been provided as individual static methods in **java.util.stream.Collectors** class. Let us now take a look at the important reduction operations already implemented in **Collectors** class and their purpose.

| **Reduction Operation(s)** | **Method(s)** | **Purpose** |
| --- | --- | --- |
| **averaging** | **averagingDouble(), averagingLong(),**[averagingInt()](https://www.javabrahman.com/java-8/java-8-how-to-use-collectors-averagingint-averaginglong-averagingdouble-with-examples/) | **To average elements of type Double/Long/Integer after applying a mapping function to the elements to extract respective values to be averaged** |
| **counting** | [counting()](https://www.javabrahman.com/java-8/java-8-counting-with-collectors-collectors-counting-method-tutorial-with-examples/) | **Count the number of stream elements** |
| **grouping** | [groupingBy()](https://www.javabrahman.com/java-8/java-8-grouping-with-collectors-groupingby-method-tutorial-with-examples/) | **To produce Map of elements grouped by grouping criteria provided** |
| **String concatenation** | [joining()](https://www.javabrahman.com/java-8/java-8-joining-with-collectors-collectors-joining-method-tutorial-with-examples/) | **For concatenation of stream elements into a single String** |
| **mapping** | [mapping()](https://www.javabrahman.com/java-8/java-8-how-to-use-collectors-mapping-collector-with-examples/) | **Applying a mapping operation to all stream elements being collected** |
| **minimum and maximum determination** | [minBy()/maxBy()](https://www.javabrahman.com/java-8/java-8-finding-maxmin-with-collectors-maxby-minby-methods-tutorial-with-examples/) | **To find minimum/maximum of all stream elements based on Comparator provided** |
| **partitioning** | [partitioningBy()](https://www.javabrahman.com/java-8/java-8-partitioning-with-collectors-partitioningby-method-tutorial-with-examples/) | **To partition stream elements into a Map based on the Predicate provided** |
| **reduction** | **reducing()** | **Reducing elements of stream based on BinaryOperator function provided** |
| **summarization** | **summarizingDouble(), summarizingLong(),**[summarizingInt()](https://www.javabrahman.com/java-8/java-8-how-to-use-collectors-summarizingint-summarizinglong-summarizingdouble-with-examples/) | **To summarize stream elements after mapping them to Double/Long/Integer value using specific type Function** |
| **summation** | **summingDouble(), summingLong(), summingInt()** | **To sum-up stream elements after mapping them to Double/Long/Integer value using specific type Function** |
| **collect into a Collection** | [toCollection()](https://www.javabrahman.com/java-8/java-8-how-to-use-collectors-tocollection-collector-with-examples/) | **To collect stream elements into a collection** |
| **collect into a Map/ConcurrentMap** | **toMap()/toConcurrentMap()** | **To collect stream elements into a map/concurrent map after applying provided key/value determination Function instances to the elements** |
| **collect in a List** | **toList()** | **Collects stream elements in a List** |
| **collect in a Set** | **toSet()** | **Collects stream elements in a Set** |
| **collect and transform** | [collectingAndThen()](https://www.javabrahman.com/java-8/java-8-how-to-use-collectors-collectingandthen-method-with-examples/) | **Collects stream elements and then transforms them using a Function** |
| **Table: Predefined collectors returned by static methods of java.util.stream.Collectors class** *(Links to individual tutorials on each Collector type is in the****Method(s)****column*) | | |

As you can see, the above list of predefined collectors is quite exhaustive and covers a wide range collector usages. As a result, most of the times your collecting needs will be fulfilled by a Collector from the list above. In rare cases, when you need to collect in a way different from those listed, you will have to implement your own custom Collector.

Let us now see how to use a predefined collector returned by method **Collectors.partitioningBy()**. Using this predefined **Collector** one can easily partition the elements of a stream into a Map with elements divided into true/false entries based on an input [Predicate](https://www.javabrahman.com/java-8/java-8-java-util-function-predicate-tutorial-with-examples/) passed to it.  
*(Note – If interested, you can read the full tutorial on partitioningBy Collector*[*here*](https://www.javabrahman.com/java-8/java-8-partitioning-with-collectors-partitioningby-method-tutorial-with-examples/)*)*

**Java 8 Code example showing pre-defined Collector in action**

**Java 8 code showing pre-defined Collector 'Collectors.toList()' usage**

|  |
| --- |
| //BasicCollector.java  import com.javabrahman.java8.Employee;    import java.util.Arrays;  import java.util.List;  import java.util.Map;  import java.util.stream.Collectors;    public class BasicCollector {    static List<Employee> employeeList = Arrays.asList(new Employee("Tom Jones", 45),        new Employee("Harry Major", 25),        new Employee("Ethan Hardy", 65),        new Employee("Nancy Smith", 22),        new Employee("Deborah Sprightly", 29));      public static void main(String args[]){      Map<Boolean,List<Employee>> employeeMap =        employeeList.stream()                    .collect(Collectors.partitioningBy((Employee emp) -> emp.getAge() > 30));      System.out.println("Employees partitioned based on age");      employeeMap.forEach((Boolean key, List<Employee> empList) -> System.out.println(key +"->" + empList));    }    }    //Employee.java(POJO)  package com.javabrahman.java8;  public class Employee {    private String name;    private Integer age;      public Employee(String name, Integer age) {      this.name = name;      this.age = age;    }      //Setters & Getters for 'name' and 'age' go here      public String toString(){      return "Employee Name:"+this.name          +"  Age:"+this.age;     }      @Override    public boolean equals(Object obj) {      if (obj == this) {        return true;      }      if (!(obj instanceof Employee)) {        return false;      }      Employee empObj = (Employee) obj;      return this.age == empObj.age          && this.name.equalsIgnoreCase(empObj.name);    }      @Override    public int hashCode() {      int hash = 1;      hash = hash \* 17 + this.name.hashCode();      hash = hash \* 31 + this.age;      return hash;    }  } |

**OUTPUT of the above code**

Employees partitioned based on age  
false->[Employee Name:Harry Major Age:25, Employee Name:Nancy Smith Age:22, Employee Name:Deborah Sprightly Age:29]

true->[Employee Name:Tom Jones Age:45, Employee Name:Ethan Hardy Age:65]

**Explanation of the code**

* **Employee** is the POJO class for the above example, which contains 2 attributes – **name** and **age**.
* In the **main()** method of **BasicCollector** class we first create a **List** of 5 employees, named **employeeList**, using [Arrays.asList()](https://www.javabrahman.com/corejava/converting-an-array-to-listor-arraylist-in-java-options-and-their-analysis/) method.
* Next we create a stream of elements of **employeeList** using **employeeList.stream()** method.
* We then collect this stream of employee objects using the [pipelined](https://www.javabrahman.com/programming-principles/pipelines-in-computing-and-software-engineering/) **collect()** method to which we pass an instance of **Collector** returned by **Collectors.partitioningBy()** method. We partition the employees based on whether their **age > 30** or not. Accordingly, we pass a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) for this predicate as input to the **partitioningBy()** method.
* We then print the entries for **employeeMap** returned by the **collect()** method using **Map.forEach()**method. The output is as expected – a **Map** containing two entries for keys – **true** and **false**, and the values contains lists of **Employee** objects satisfying/not satisfying the predicate passed.

**Summary and further articles in the Java 8 Collector Series**  
This tutorial was an introduction to the basics of **Collector** interface, the components of a Collector and how these components act in cohesion to collect the stream elements into a final collection. We also had an overview of the predefined collectors defined in **java.util.stream.Collectors** class and saw a code example showing a predefined collector in action.

The stage is now set for us to delve deeper into the concepts of collectors. To begin with, in the next article of Collectors series, we will explore how a Collector can perform its duty of collecting stream elements in parallel to improve performance. This will be followed by detailed individual tutorials dedicated to each of the predefined Collectors we saw briefly above. The Collectors series will finally culminate with a tutorial explaining how to create a custom collector of your own.

**Java 8 Grouping with Collectors | groupingBy method tutorial with examples**

[March 24, 2017](https://www.javabrahman.com/java-8/java-8-grouping-with-collectors-groupingby-method-tutorial-with-examples/)

**Introduction** – Java 8 Grouping with Collectors tutorial explains how to use the predefined Collector returned by **groupingBy()** method of **java.util.stream.Collectors** class with examples.

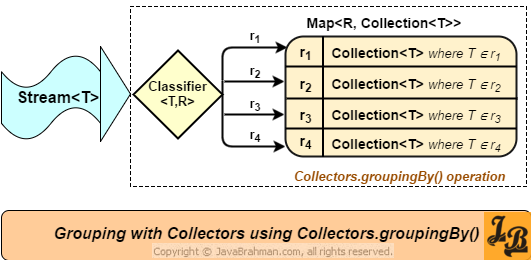
The tutorial begins with explaining how grouping of stream elements works using a *Grouping Collector*. The concept of grouping is visually illustrated with a diagram. Next, the three overloaded **groupingBy()** methods in **Collectors** class are explained using their method definitions, Java code examples showing the 3 methods in action and explanations for the code examples. Lastly, a brief overview of the concurrent versions of the three **groupingBy()** methods is provided.  
*(Note – This tutorial assumes that its readers are familiar with the basics of*[*Java 8 Collectors*](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/)*.)*

**Understanding the concept of ‘grouping’ using Collectors**  
Given a stream of objects, there are scenarios where these objects need to be grouped based on a certain distinguishing characteristic they posses. This concept of grouping is the same as the ‘group by’ clause in SQL which takes an attribute, or a calculated value derived from attribute(s), to divide the retrieved records in distinct groups. Generally, in imperative style of programming, such grouping of records(objects in OOPS) involves iterating over each object, checking which group the object being examined falls in, and then adding that object in its correct group. The group itself is held together using a **Collection** instance. Java 8’s new functional features allow us to do the same grouping of objects in a declarative way, which is typical of functional rather than [imperative](https://www.javabrahman.com/programming-principles/imperative-versus-functional-programming-paradigms-conceptual/) style of programming, using Java 8’s new *Grouping Collector*.

Grouping collectors use a *classification function*, which is an instance of the Function<T,R> functional interface, which for every object of type **T** in a stream, returns a classifier object of type **R**. Various values of R, finite in number, are the ‘group names’ or ‘group keys’. As the grouping collector works on the stream of objects its collecting from it creates collections of stream objects corresponding to each of the ‘group keys’. I.e. for every value of **R** there is a collection of objects all of which return that value of **R** when subjected to the classification function.

All these R-values and corresponding Collection of stream objects are stored by the grouping collector in a **Map<R, Collection<T>>**, i.e. each **‘key,value’** entry in the map consists of **‘R,Collection<T>’**.

The process of grouping, starting from the application of classification function on the stream elements, till the creation of Map containing the grouped elements, is as shown in the diagram below –



In the above diagram, the elements of **Stream<T>** are grouped using a classification function returning 4 values of **R** – **r1,r2,r3,r4**. The grouped elements are stored in a **Map<R,Collection<T>>**, with the 4 values of **R** being used as 4 *keys* pointing to 4 corresponding **collections** stored in the **Map**. These **Collection** instances hold the individual grouped elements, which is the required output from the grouping collector.

Having understood now the concept of grouping with collectors, let us now see how to implement grouping collectors in code using the 3 overloaded **groupingBy()** method variants provided in **Collectors** class, starting from the simplest variant which creates a **List** of the grouped elements.

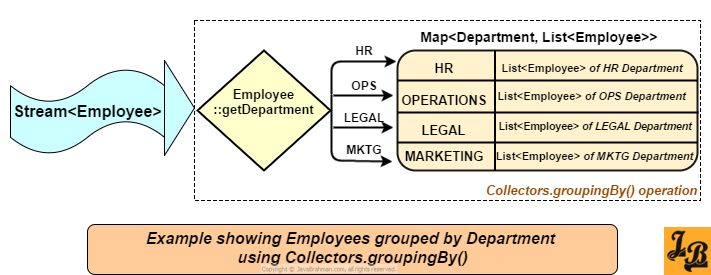
**Variant #1 of Collectors.groupingBy() method – stores grouped elements in a List**  
The simplest of **Collectors.groupingBy()** method variants is defined with the following signature –

**public static <T, K> Collector<T, ?, Map<K, List<T>>> groupingBy(Function<? super T, ? extends K> classifier)**

Where,  
     – input is **classifier** which is an instance of a [Function](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/) functional interface which converts from type T to type K.  
     – output is a Collector with [finisher](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples#ref1)(return type) as a **Map** with entries having ‘**key,value**’ pairs as ‘**K, List<T>**’

The simplest variant of **groupingBy()** method applies classifier **Function<T,R>** to each individual element of type **T** collected from **Stream<T>**. It then groups elements into individual lists based on the value of **R** they return on application of classifier function, and stores them in a **Map<R,List<T>>**, using the process we had understood in the previous section explaining how a *grouping collector* operates.

**Variant #1 of grouping collector – Java Example**  
Lets say we have a stream of **Employee** objects, belonging to a company, who need to be grouped by their departments, with their **Department** present as an attribute in the **Employee** object. As the end result of applying the grouping collector for achieving this we want a **Map** with keys as departments and corresponding values as **List** of employees in that department. Diagrammatically such as an implementation would be represented as shown below –



In the above diagram, employees are grouped into 4 departments – HR, OPERATIONS, LEGAL and MARKETING. Let us now see the Java code for implementing the above ‘Department – Employees’ use case, followed by its explanation.

**Java 8 code example for Variant #1 of Collectors.groupingBy()**

|  |
| --- |
| package com.javabrahman.java8.collector;  import java.util.Arrays;  import java.util.List;  import java.util.Map;  import java.util.stream.Collectors;    public class GroupingWithCollectors {    static List<Employee> employeeList = Arrays.asList(        new Employee("Tom Jones", 45, 12000.00,Department.MARKETING),        new Employee("Harry Major", 26, 20000.00, Department.LEGAL),        new Employee("Ethan Hardy", 65, 30000.00, Department.LEGAL),        new Employee("Nancy Smith", 22, 15000.00, Department.MARKETING),        new Employee("Catherine Jones", 21, 18000.00, Department.HR),        new Employee("James Elliot", 58, 24000.00, Department.OPERATIONS),        new Employee("Frank Anthony", 55, 32000.00, Department.MARKETING),        new Employee("Michael Reeves", 40, 45000.00, Department.OPERATIONS));    public static void main(String args[]){      Map<Department,List<Employee>> employeeMap          = employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment));      System.out.println("Employees grouped by department");      employeeMap.forEach((Department key, List<Employee> empList) -> System.out.println(key +" -> "+empList));      }  }  //Employee.java - POJO Class  package com.javabrahman.java8.collector;  public class Employee {    private String name;    private Integer age;    private Double salary;    private Department department;      public Employee(String name, Integer age, Double salary, Department department) {      this.name = name;      this.age = age;      this.salary = salary;      this.department = department;    }      // Setters/Getters for name,age,salary,department go here      public String toString(){      return "Employee Name:"+this.name;    }      //Standard equals and hashcode implementations go here    }  //Enum Department.java  package com.javabrahman.java8.collector;  public enum Department {    HR, OPERATIONS, LEGAL, MARKETING  } |

**OUTPUT of the above code**

Employees grouped by department

HR -> [Employee Name:Catherine Jones]

LEGAL -> [Employee Name:Harry Major, Employee Name:Ethan Hardy]

OPERATIONS -> [Employee Name:James Elliot, Employee Name:Michael Reeves]

MARKETING -> [Employee Name:Tom Jones, Employee Name:Nancy Smith, Employee Name:Frank Anthony]

**Explanation of the code**

* **Employee** is the POJO class in the above example of which we create a Stream. It has four attributes – **name**, **age**, **department** and **salary**.
* **Department** is an **Enum** with the following values – **HR**, **OPERATIONS**, **LEGAL**, **MARKETING**.
* **employeeList** is a static list of 8 **Employee**s.
* In the **main()** method of **GroupingWithCollectors** class we create a **Stream** of **Employee**s using the **stream()** method of **List** interface.
* On the stream of **Employee**s we call the **collect()** method with predefined **Collector** returned by **Collectors.groupingBy()** method as the parameter.
* The **classification function** passed to **groupingBy()** method is the [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/) to **Employee.getDepartment()** method specified as **"Employee::getDepartment"**.
* Lastly, the **Map** of employees grouped by department is printed using **Map.forEach()** method. The output is as expected – map contains entries of **‘key,value’**in the form of **‘Department, List<Employee>’**with an *entry* for containing a **Department** as **key** having the **List** of **Employee**s of that **Department**stored as value.

**Variant #2 of Collectors.groupingBy()- uses a user specified Collector to collect grouped elements**  
Whereas the 1st variant always returned a **List** containing the elements of a group, **the 2nd variant of grouping collector provides the flexibility to specify how the grouped elements need to be collected using a second parameter which is a Collector**. So, instead of just storing the groups in resultant **Map** as **Lists**, we can instead store them in say **Sets**, or find the maximum value in each group and store it rather than storing all the elements of a group, or any such collector operation which is applicable on the stream elements.

The 2nd variant of grouping collector is defined with the following signature –

**Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier,  
Collector<? super T, A, D> downstream)**

Where,  
     – 1st input parameter is **classifier** which is an instance of a [Function](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/) functional interface which converts from type **T** to type **K**.  
     – 2nd input parameter is **downstream** collector which collects the grouped elements into type **D**, where **D** is the specified [finisher](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples#ref1).  
     – output is a **Collector** with finisher(return type) as a **Map** with entries having ‘**key,value**’ pairs as ‘**K, D**’

**How variant#1 and variant#2 of grouping collector are closely related**  
In the Collectors class’ code, the first variant of grouping Collector which accepts just the classification function as input does not itself return the Collector which processes the Stream elements. Instead, internally it delegates the call forward to the second variant with the call – **groupingBy(classifier, toList())**. So, first variant of grouping collector is thus just a convenient way of invoking the second variant with the downstream collector ‘hardcoded’ as a **List**.

Let us now see the 2nd variant of grouping collector in action with a Java code example.

**Variant #2 of grouping collector – Java Example**  
This example for variant#2 uses the same use case of employees being grouped as per their department but this time instead of storing the grouped elements in a **List**, we will instead store them inside a **Set** in the resultant **Map**.  
*(Note – The****Employee****class and****employeeList****objects with their values remain the same as the previous code usage example and hence are not shown below for brevity.)*

**Java 8 code example for VARIANT #2 of Collectors.groupingBy()**

|  |
| --- |
| public static void main(String args[]){    Map<Department,Set<Employee>> employeeMap      = employeeList.stream()        .collect(Collectors.groupingBy(Employee::getDepartment, Collectors.toSet()));    System.out.println("Employees grouped by department");    employeeMap.forEach((Department key, Set<Employee> empSet) -> System.out.println(key +" -> "+empSet));    } |

**OUTPUT of the above code**

Employees grouped by department

HR -> [Employee Name:Catherine Jones]

LEGAL -> [Employee Name:Harry Major, Employee Name:Ethan Hardy]

OPERATIONS -> [Employee Name:James Elliot, Employee Name:Michael Reeves]

MARKETING -> [Employee Name:Tom Jones, Employee Name:Nancy Smith, Employee Name:Frank Anthony]

**Explanation of the code**

* The code above is ‘nearly’ the same as the code for 1st variant of grouping collector. The main difference is that **Collectors.grouping()** method is now passed a second parameter – **Collectors.toSet()** – which tells the grouping collector to collect the grouped values in individual **Sets**.
* The output with employees grouped in **Sets** looks the same as 1st variant’s output as individual set elements are enclosed between square brackets -‘[]’ – just like they were for **Lists**. But, if you look closely at the code then you will find that the **employeeMap.forEach()** method call now has a **Set<Employee>** specified as the type of **value** rather than a **List** which was the case in the 1st variant.

**Variant #3 of Collectors.groupingBy()- with user specified Supplier function for Map creation and Collector to collect grouped elements**  
Whereas the 1st variant always returned a **List** containing the elements of a group, the 2nd variant of grouping collector provides the flexibility to specify how the grouped elements need to be collected, **the 3rd variant adds the capability to specify how the Map which holds the result is created**. So, using the 3rd variant of grouping **Collector** it can be specified whether the resultant **Map** containing the grouped values is a **HashMap** or a **TreeMap**, or some user specified type of **Map**.

The 3rd variant of grouping collector is defined with the following signature –

**Collector<T, ?, M> groupingBy(Function<? super T, ? extends K> classifier, Supplier<M> mapFactory, Collector<? super T, A, D> downstream)**

Where,  
     – 1st input parameter is **classifier** which is an instance of a [Function](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/) functional interface which converts from type **T** to type **K**.  
     – 2nd input parameter is [Supplier<M>](https://www.javabrahman.com/java-8/java-8-java-util-function-supplier-tutorial-with-examples/) which is a [factory](https://www.javabrahman.com/design-patterns/factory-method-design-pattern-in-java/) supplying **Maps** of type **M**.  
     – 3rd input parameter is **downstream** collector which collects the grouped elements into type **D**, where **D** is the specified [finisher](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples#ref1).  
     – output is a **Collector** with finisher(return type) as a **Map** with entries having ‘**key,value**’ pairs as ‘**K, D**’

**How variant#2 and variant#3 of grouping collector are closely related**  
In the Collectors class’ code, the second variant of grouping Collector which accepts the classification function along with downstream collector as input does not itself return the collector which processes the stream elements. Instead, internally it delegates the call forward to the third variant with the call – **groupingBy(classifier, HashMap::new, downstream);**. So, second variant of grouping collector is thus just a convenient way of invoking the third variant with the ***Map****factory****Supplier*** ‘hardcoded’ as **HashMap::new**.

Going back a bit, we said something similar about the first and second **groupingBy()** variants as well. **Thus, we actually have a transitive kind of relationship between the three variants. Variant #1 calls variant #2 with downstream collector hardcoded, and variant #2 calls variant #3 with Map Supplier factory hardcoded.**Inferring transitively, we can now say that variant #1 actually calls variant #3 with both the downstream collector and Map Supplier factory hardcoded.

Fortunately, the transitive offloading/delegation between variants ends at variant #3 which actually contains the entire collector logic for a grouping collector.

Let us now see a Java code example showing how to use the 3rd variant of grouping collector.

**Variant #3 of grouping collector – Java Example**  
This example for variant #3 uses the same use case of employees being grouped as per their department. However, this time we will store the grouped elements in a **Set** and tell the grouping collector to store the grouped employees in a **TreeMap** instance instead of the default **HashMap** instance that was internally hardcoded in variant #2.  
*(Note – The****Employee****class and****employeeList****objects with their values remain the same as the previous code usage example and hence are not shown below for brevity.)*

**Java 8 code example for VARIANT #3 of Collectors.groupingBy()**

|  |
| --- |
| public static void main(String args[]){    Map<Department,Set<Employee>> employeeMap      = employeeList.stream()        .collect(Collectors.groupingBy(Employee::getDepartment, TreeMap::new, Collectors.toSet()));    System.out.println("Employees grouped by department");    employeeMap.forEach((Department key, Set<Employee> empSet) -> System.out.println(key +" -> "+empSet));    } |

**OUTPUT of the above code**

Employees grouped by department

HR -> [Employee Name:Catherine Jones]

OPERATIONS -> [Employee Name:James Elliot, Employee Name:Michael Reeves]

LEGAL -> [Employee Name:Harry Major, Employee Name:Ethan Hardy]

MARKETING -> [Employee Name:Tom Jones, Employee Name:Nancy Smith, Employee Name:Frank Anthony]

**Explanation of the code**

* The code above is ‘nearly’ the same as the code for 2nd variant of grouping collector. The main difference is that **Collectors.grouping()** method is now passed a third parameter as well – **TreeMap::new()** – which tells the grouping collector to collect the grouped values in an instance of a **TreeMap**.
* The output with employees grouped in **Sets** corresponding to their departments is similar to what we saw in the java examples for 1st and 2nd variants. However, this time the department names, which are the keys of the result **Map**, are arranged in alphabetical order which was not the case in the previous outputs. This alphabetical ordering is because of the use of **TreeMap** this time which automatically sorts its entries based on the natural ordering of its keys.

**Concurrent versions of grouping collector**  
We saw three **groupingBy()** method variants above which are good but not optimized for concurrent execution. In case you want to execute grouping collectors in a concurrent manner in a multi-threaded execution environment, then you can utilize the three overloaded methods in **java.util.stream.Collectors** class all of whom are named **groupingByConcurrent()**. These three concurrent methods have exactly the same signature as their non-concurrent counterparts – the same input parameters and the same return types respectively – their usage, apart from being used in concurrent contexts, is exactly the same as described above.

**Conclusion**  
In the above tutorial we understood what the concept of grouping in the context of collectors entails, looked at the three grouping collector variants, understood their definition and working in depth using diagrams, code examples, and then saw how the three variants of **groupingBy()** methods are closely interlinked. Lastly, we touched upon the concurrent grouping by collectors as well.

**Java 8 Partitioning with Collectors | partitioningBy method tutorial with examples**

[February 24, 2017](https://www.javabrahman.com/java-8/java-8-partitioning-with-collectors-partitioningby-method-tutorial-with-examples/)

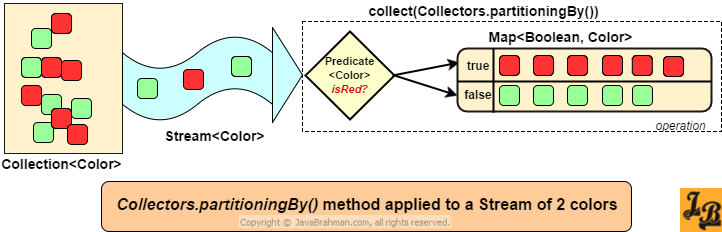
**Introduction** – Java 8 Partitioning with Collectors tutorial explains how to use the predefined Collector returned by **partitioningBy()** method of **java.util.stream.Collectors** class with examples. The tutorial starts off with explaining the concept of partitioning data in Streams with a visual example. It then discusses the advantage that partitioning Streams with Collectors provides over filtering. The partitioningBy() method is then discussed and its usage is shown with a code example. Next, we will take a look at the 2nd variant of **partitioningBy()** method, by extending the visual example we saw earlier, to see how a Collector can be used as to *again* collect the data returned by the application of **partitioningBy()** method. Lastly, we will see a Java code example showing the overloaded partitioningBy() method with a second Collector in action.  
*(Note – This tutorial assumes that you are familiar with basics of*[*Java 8 Collectors*](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/)*.)*

**Understanding the concept of ‘partitioning’ using Collectors**  
Given a stream of objects, many-a-times we need to check whether object(s) in the given stream match a specific criteria or not. Instead of writing logic for iterating over the stream elements and checking each object whether it matches the criteria (which is more of an [imperative rather than functional](https://www.javabrahman.com/programming-principles/imperative-versus-functional-programming-paradigms-conceptual/) style of programming), Java 8 Collectors allow declarative partitioning of elements into 2 groups which satisfy/don’t satisfy the given [Predicate](https://www.javabrahman.com/java-8/java-8-java-util-function-predicate-tutorial-with-examples/) of type **T**.

**Example explaining the basic concept of partitioning**  
Suppose you have a collection of blocks. These blocks are in 2 colors – green and red. Now, you want to partition the blocks into their separate color-coded groups. I.e. one collection of green blocks and another collection of red blocks.

Since, our objective is to solve this partitioning problem programmatically, hence we create a representation of this problem in Java. We define an element Color which represents the block, i.e. Color green represents green blocks and likewise Color red represents red blocks. We then create a Stream of these Color objects and use the **Collectors.partitioningBy()** method to partition these objects into 2 lists – one for each color.

This is how the above scenario would look like when drawn as a diagram –



As shown in above diagram, application of **Collectors.partitioningBy()** method to the Stream of Color objects, with predicate condition as ‘**Color.isRed()**’, results in 2 separate lists of objects being created. These lists are in 2 separate entries in a Map. The objects which return true for ‘**Color.isRed()**’, i.e. Red Color objects, are stored in the Map entry with key ‘**true**’. Similarly, the remaining green objects which return false for ‘**Color.isRed()**’ condition are store in the Map entry with key ‘**false**’.

The Color objects are thus partitioned into 2 Lists which can be retrieved by invoking **Map.get(true)** and **Map.get(false)** respectively.

**Advantage of partitioning using Collectors versus the Stream.filter() operation**  
If you are aware of the [Stream.filter()](https://www.javabrahman.com/java-8/java-8-filtering-and-slicing-streams-tutorial-with-examples/) operation then you would have realized by now that the same conditional fetching of objects based on a provided **Predicate** can be accomplished by filtering stream elements as well. However, the partitioning operation provides a simple but helpful advantage over filtering. At the end of the partitioning operation, the method returns back both the groups of elements – one that satisfy the given **Predicate** and the ones that don’t- *together*. Filtering a stream can provide you the same two groups but you will need to invoke the filtering operation twice – one with the given Predicate and the second time with the negation of that Predicate.

Now that we have seen how partitioning with collectors works, its time to see how to implement partitioning in code using the predefined Collector instance returned by the static method **Collectors.partitioningBy()**.

**Two overloaded variants of Collectors.partitioningBy() method**  
At this point it is important to note that there are actually 2 overloaded static methods named **partitioningBy()** in the **Collectors** class. What we are looking at now is the first of these methods which accepts a **Predicate** instance as its only parameter. There is a second overloaded **Collectors.partitioningBy()** method as well which along with a **Predicate** instance takes another **Collector** instance as the second input parameter. We will look at the second **partitioningBy()** method also in detail after we cover the first one.

**Collectors.partitioningBy() method**  
**Collectors.partitioningBy()** method is defined with the following signature –

**Collector<T, ?, Map<Boolean, List<T>>> partitioningBy(Predicate<? super T> predicate)**

Where,  
     – input is **predicate** which is an instance of a [Predicate](https://www.javabrahman.com/java-8/java-8-java-util-function-predicate-tutorial-with-examples/) Functional Interface of type **T**  
     – output is a Collector with [finisher](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples#ref1)(return type) as a **Map** with entries having ‘**key,value**’ pairs as ‘**Boolean, List<T>**’

When the **Stream.collect()** operation is invoked on a **Stream** containing elements of type **T**, with **Collector<T>** returned by **Collectors.partitioningBy(Predicate<T>)** method passed as parameter, what you get as the resultant collection from this [terminal operation](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/) is a **Map** containing the elements of the **Stream** divided into two entries(or ‘key,value’ pairs). While the 1st map entry has key **true** and value containing **List<T>** of elements that satisfy the **Predicate** condition, the 2nd map entry has key **false** and value containing **List<T>** of elements which *do not* satisfy the **Predicate** condition.

Let us see a Java 8 code example showing the **Collector** returned by **Collectors.partitioningBy()**method in action.

**Java 8 code showing Collectors.partitioningBy() usage**

**package com.javabrahman.java8.collector;**

**import com.javabrahman.java8.Employee;**

**import java.util.Arrays;**

**import java.util.List;**

**import java.util.Map;**

**import java.util.stream.Collectors;**

**public class PartitioningWithCollectors {**

**static List<Employee> employeeList = Arrays.asList(new Employee("Tom Jones", 45),**

**new Employee("Harry Major", 26),**

**new Employee("Ethan Hardy", 65),**

**new Employee("Nancy Smith", 22),**

**new Employee("Catherine Jones", 21),**

**new Employee("James Elliot", 58),**

**new Employee("Frank Anthony", 55),**

**new Employee("Michael Reeves", 40));**

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

**Map<Boolean,List<Employee>> employeeMap**

**= employeeList**

**.stream()**

**.collect(Collectors.partitioningBy((Employee emp) -> emp.getAge() > 30));**

**System.out.println("Employees partitioned based on Predicate - 'age > 30'");**

**employeeMap.forEach((Boolean key, List<Employee> empList) -> System.out.println(key +"->" + empList));**

**}**

**}**

**//Employee.java(POJO Class)**

**package com.javabrahman.java8;**

**public class Employee {**

**private String name;**

**private Integer age;**

**public Employee(String name, Integer age) {**

**this.name = name;**

**this.age = age;**

**}**

**//Getters and Setters of name & age go here**

**public String toString(){**

**return "Employee Name:"+this.name**

**+" Age:"+this.age;**

**}**

**@Override**

**public boolean equals(Object obj) {**

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

**return true;**

**}**

**if (!(obj instanceof Employee)) {**

**return false;**

**}**

**Employee empObj = (Employee) obj;**

**return this.age == empObj.age**

**&& this.name.equalsIgnoreCase(empObj.name);**

**}**

**@Override**

**public int hashCode() {**

**int hash = 1;**

**hash = hash \* 17 + this.name.hashCode();**

**hash = hash \* 31 + this.age;**

**return hash;**

**}**

**}**

**OUTPUT of the above code**

Employees partitioned based on Predicate – ‘age > 30’  
false->[Employee Name:Harry Major Age:26, Employee Name:Nancy Smith Age:22, Employee Name:Catherine Jones Age:21]

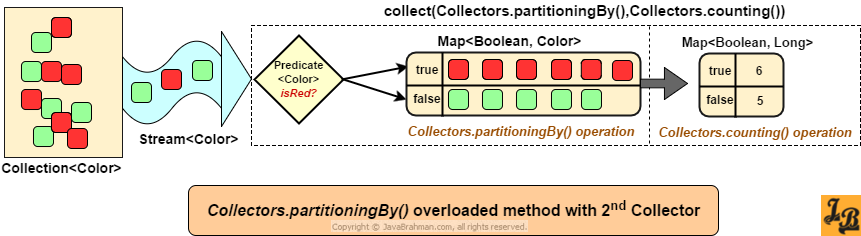
true->[Employee Name:Tom Jones Age:45, Employee Name:Ethan Hardy Age:65, Employee Name:James Elliot Age:58, Employee Name:Frank Anthony Age:55, Employee Name:Michael Reeves Age:40]

**Explanation of the code**

* **Employee** is the POJO class in the above example of which we create a Stream. It has two main attributes – **name**and **age**.
* **employeeList** is a static list of 8 **Employee**s.
* In the **main()** method of **PartitioningWithCollectors** class we create a **Stream** of **Employee**s using the **stream()** method of **List** interface.
* On the stream of Employees we call the **collect()** method with the **Predicate** instance being specified as its equivalent [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) – **(Employee emp) -> emp.getAge()>30)**. This predicate condition states that the employee’s age should be greater than 30 years.
* Lastly, the **Map** of employees partitioned by the predicate condition are printed using **Map.forEach()**method. The output is as expected – employees with **age>30** are printed in a list corresponding to key value **true**, while those with **age<=30** are printed as a list against key value **false**.

**Overloaded Collectors.partitioningBy() with Collector as second parameter**  
To understand the utility and usage of the overloaded **partitioningBy()** method, let us revisit the earlier example where we partitioned the collection of Color objects into red and green lists. However, what if your requirement was not the partitioned lists but instead you needed a count of red and green color objects as the final result of partitioning. Using a 2nd **Collector** in this case, specifically the one returned by **Collectors.counting()** method, is exactly what you need to get the count of each of these lists.

Now, have a look at the diagram below which extends the previous visual Colors example by calling the overloaded **partitioningBy()** method with the counting collector -



In the above diagram, output from the **partitioningBy()** method is a map with values containing the count of red and green colors. In fact, the lists of colors were created by the **partitioningBy()** method in this case as well, but then the counting collector was applied on the lists and a **Map** was returned which had just the count of colors as the value for keys **true** and **false**.

Having understood the working of the overloaded **partitioningBy()** method, let us now take a look at its formal definition -

**Collector<T, ?, Map<Boolean, D>> partitioningBy(Predicate<? super T> predicate,  
Collector<? super T, A, D> downstream)**

Where,  
     - first parameter is **predicate** which is an instance of a Predicate Functional Interface  
     - second parameter is a **Collector**  
     - output is a Collector with finisher(return type) as a**Map** with entries having ‘**key,value**’ pairs as ‘**Boolean, D>**’ where **D** is the return type of the finisher function of second collector parameter

Let us extend the previous code example, where we partitioned the employees into 2 groups based on whether they were older than 30 years or not, and pass the **Collector** returned by **Collectors.counting()** method as the overloaded **partitionBy()** method’s second parameter.  
*(Note - The****Employee****class and****employeeList****objects with their values remain the same as the previous code usage example and hence are not shown below for brevity.)*

**Java 8 code showing Collectors.partitioningBy() method usage**

**Map<Boolean,Long> employeeMapCount =**

**employeeList.stream()**

**.collect(Collectors.partitioningBy(**

**(Employee emp) -> (emp.getAge() > 30),**

**Collectors.counting()**

**));**

**System.out.println("Employee count in the 2 partitioned age groups");**

**employeeMapCount.forEach((Boolean key,Long count) -> System.out.println(key +" count -> "+ count));**

**OUTPUT of the above code**

Employee count in the 2 partitioned age groups  
false count -> 3  
true count -> 5

**Explanation of the code**

* **Collectors.partitioningBy()** is invoked with **Predicate** lambda being same as earlier i.e. **(Employee emp) -> emp.getAge()>30)**.
* The second parameter to the **partitioningBy()** method is the **Collector** returned by **Collectors.counting()** method.
* As expected, a **Map** of values is returned, named **employeeMapCount**,which when printed using **Map.forEach()** method gives the count of employees in the 2 partitioned groups as **3** and **5** respectively for **true** and **false** keys.

**Conclusion** - In this tutorial we first understood what is meant by partitioning of Streams using a **Collector**along with its advantage over filtering using **Stream.filter()** method. We then understood the working of predefined **Collector** returned by **Collectors.partitioningBy()** method with a visual example followed by a Java 8 code example. We then understood the working of the overloaded partitioning method with a second collector by extending the previous visual and code examples.

**Java 8 Counting with Collectors | Collectors.counting method tutorial with examples**

[February 25, 2017](https://www.javabrahman.com/java-8/java-8-counting-with-collectors-collectors-counting-method-tutorial-with-examples/)

Java 8 Counting with Collectors tutorial explains, with examples, how to use the predefined **Collector**returned by **java.util.Stream.Collectors** class’ **counting()** method to count the number of elements in a **Stream**. The tutorial starts off with the formal definition of the **Collectors.counting()** method, then explains its working, and finally shows the method’s usage via a Java code example.  
*(Note – This tutorial assumes that you are familiar with basics of*[*Java 8 Collectors*](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/)*.)*

**Predefined Collector returned by Collectors.counting() method**  
**Collectors.counting()** method is defined with the following signature –

**public static <T> Collector<T, ?, Long> counting()**

Where,  
     – output is a **Collector**, acting on a Stream of elements of type **T**, with its [finisher](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples#ref1) returning the ‘collected’ value of type **Long**

The working of the counting collector is pretty straightforward. It is a [terminal operation](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/) which returns the total count of elements in the stream which reach the **collect()** method after undergoing various [pipelined](https://www.javabrahman.com/programming-principles/pipelines-in-computing-and-software-engineering/) stream operations such as filtering, reduction etc.

It’s also worthwhile to know that internally the counting collector actually delegates the counting job to the ‘reducing collector’ with the call **reducing(0L, e -> 1L, Long::sum)**. *But this call is internal to the implementation and need not be taken into consideration before terminating your stream with the counting Collector.*

**Java 8 code example showing Collector.counting() method’s usage**

**Java 8 code example showing Collectors.counting() usage**

|  |
| --- |
| package com.javabrahman.java8.collector;  import com.javabrahman.java8.Employee;  import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;  public class CountingWithCollectors {    static List<Employee> employeeList = Arrays.asList(new Employee("Tom Jones", 45),        new Employee("Harry Major", 25),        new Employee("Ethan Hardy", 65),        new Employee("Nancy Smith", 22),        new Employee("Deborah Sprightly", 29));      public static void main(String args[]){      Long count=employeeList.stream().collect(Collectors.counting());      System.out.println("Employee count: "+count);    }  }  //Employee.java(POJO Class)  package com.javabrahman.java8;  public class Employee {    private String name;    private Integer age;      public Employee(String name, Integer age) {      this.name = name;      this.age = age;    }      //Getters and Setters of name & age go here    public String toString(){      return "Employee Name:"+this.name          +"  Age:"+this.age;    }    //Standard equals() and hashcode() implementations go here  } |

**OUTPUT of the above code**

Employee count: 5

**Explanation of the code**

* **Employee.java** is the POJO class i.e. the type of which the Stream is created.
* **CountingWithCollectors** class contains a static list of 5 **Employee** objects named **employeeList**.
* In the **main()** method of **CountingWithCollectors** class a stream of **Employee** objects is created by invoking **employeeList.stream()**. This stream is then terminated by invoking the **collect()** method with the **Collector** returned by the **Collectors.counting()** method passed as a parameter.
* As expected, the Stream terminates and the counting **Collector** returns the count of the stream elements(employees) as a **Long** value.
* Lastly, the employee count is printed and is **5** as expected.

**Summary**  
In this tutorial we understood the working of the predefined **Collector** returned by the **Collectors.counting()** method. We looked at the formal definition of the **Collectors.counting()**method, had a brief look at what actually happens inside the method, and finally saw the **Collectors.counting()** method in action with a Java 8 code example and its explanation.

**Java 8 Finding max/min with Collectors | maxBy, minBy methods tutorial with examples**

[February 26, 2017](https://www.javabrahman.com/java-8/java-8-finding-maxmin-with-collectors-maxby-minby-methods-tutorial-with-examples/)

Java 8 Determining maximum and minimum with Collectors tutorial explains, with examples, how to use the predefined **Collectors** returned by **java.util.Stream.Collectors** class’ **maxBy()** and **minBy()** methods to find the maximum and minimum element of a given **Stream**. The tutorial starts off with the formal definition of the **maxBy(), minBy()** methods, then explains their working, and finally shows the methods’ usage via a Java code example.  
*(Note – This tutorial assumes that you are familiar with basics of*[*Java 8 Collectors*](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/)*.)*

**Predefined Collectors returned by Collectors.maxBy()/Collectors.minBy() methods**  
**Collectors.maxBy()/minBy()** methods are defined with the following signatures –

**public static <T> Collector<T, ?, Optional<T>> maxBy(Comparator<? super T> comparator)**  
***AND***  
**public static <T> Collector<T, ?, Optional<T>> minBy(Comparator<? super T> comparator)**

Where,  
     – input parameter is an instance of **Comparator** of type **T**  
     – output is a **Collector**, acting on a Stream of elements of type **T**, with its [finisher](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples#ref1) returning the maximum*(for****maxBy()****)* or minimum*(for****minBy()****)* value from all the elements of the stream as an **Optional** value

The predefined collectors returned by **Collectors** class’ **maxBy()**/**minBy()** methods get the maximum/minimum elements in the stream on which they are invoked using the **Stream.collect()** method. The max/min value is returned as an **Optional** value. This is because in case the stream has no elements then to avoid sending a *bare* null value which can result in a **NullPointerException**, the value is sent as an **Optional**.

**Collectors.maxBy()**/**minBy()** methods are [terminal operations](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/).

Internally, these collectors delegate their tasks to the ‘reducing collector’ with the calls **reducing(BinaryOperator. maxBy(comparator))** and **reducing(BinaryOperator. minBy(comparator))** for maximum/minimum element determination respectively. *(Note that these calls to the reducing collector are internal to the implementation of the****Collectors****class, and need not be a point of consideration for the end programmer using these methods.)*

**Java 8 code example showing Collectors.maxBy(),minBy() methods’ usage**

**Java 8 code example showing Collectors.maxBy()/minBy() usage**

|  |
| --- |
| package com.javabrahman.java8.collector;    import com.javabrahman.java8.Employee;  import java.util.Arrays;  import java.util.Comparator;  import java.util.List;  import java.util.Optional;  import java.util.stream.Collectors;    class MaxMinWithCollectors {    static List<Employee> employeeList          = Arrays.asList(new Employee("Tom Jones", 45, 15000.00),            new Employee("Tom Jones", 45, 7000.00),            new Employee("Ethan Hardy", 65, 8000.00),            new Employee("Nancy Smith", 22, 10000.00),            new Employee("Deborah Sprightly", 29, 9000.00));      public static void main(String[] args) {      Optional<Employee> maxSalaryEmp =              employeeList.stream()              .collect(Collectors.maxBy(Comparator.comparing(Employee::getSalary)));      System.out.println("Employee with max salary:"              + (maxSalaryEmp.isPresent()? maxSalaryEmp.get():"Not Applicable"));      Optional<Employee> minAgeEmp =              employeeList.stream()              .collect(Collectors.minBy(Comparator.comparing(Employee::getAge)));      System.out.println("Employee with min age:"              + (minAgeEmp.isPresent()? minAgeEmp.get():"Not Applicable"));    }  }  //Employee.java(POJO Class)  package com.javabrahman.java8;  import java.text.DecimalFormat;  public class Employee {    private String name;    private Integer age;    private Double salary;      public Employee(String name, Integer age, Double salary) {      this.name = name;      this.age = age;      this.salary = salary;    }      //Getters and Setters of name, age & salary go here      public String toString(){      DecimalFormat dformat = new DecimalFormat(".##");      return "Employee Name:"+this.name          + "  Age:"+this.age          + "  Salary:"+dformat.format(this.salary);    }    //Standard equals() and hashcode() implementations go here  } |

**OUTPUT of the above code**

Employee with max salary:  Employee Name:Tom Jones   Age:45   Salary:15000.0  
Employee with min age:  Employee Name:Nancy Smith   Age:22   Salary:10000.0

**Explanation of the code**

* **Employee.java** is the POJO class i.e. the type of which the **Stream** is created. It contains 3 attributes – **name**, **age** and **salary**.
* **MaxMinWithCollectors** class contains a static list of 5 **Employee** objects named **employeeList**.
* In the **main()** method of **MaxMinWithCollectors** class a stream of **Employee** objects is created by invoking **employeeList.stream()**. This stream is then terminated by invoking the **collect()** method with the **Collector** returned by the **Collectors.maxBy()** method passed as a parameter.
* **Collectors.maxBy()** method needs as its input parameter a **Comparator** instance. This Comparator instance is created using the [Comparator.comparing()](https://www.javabrahman.com/java-8/the-complete-java-8-comparator-tutorial-with-examples/) method with the sort key specified using the [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/) to the getter of **salary** attribute of **Employee** class i.e. “**Employee::getSalary**”.
* As expected, the **Collector** returned by **Collectors.maxBy()** method finds the **Employee** with the maximum salary and prints it as “**Employee Name:Tom Jones Age:45 Salary:15000.0**”. Employee details are printed as per the formatting defined in the overridden **toString()** method in the **Employee**POJO class. As the maximum value is inside an **Optional**, the code uses a *ternary operator* to first determine whether the value is present and then retrieves it. Else it returns the **String** “Not Applicable”.
* Similar to the maximum salary determination, employee with minimum age is identified by first creating a stream of **Employee** objects using **employeeList.stream()** method. **Stream.collect()** method is [pipelined](https://www.javabrahman.com/programming-principles/pipelines-in-computing-and-software-engineering/) to the **stream()** method, with **Collectors.minBy()** as parameter. **Collectors.minBy()** in turn takes a **Comparator** instance defined using **Comparator.comparing()** method with getter to **Employee**’s **age** attribute passed as sort key to it via the method reference “**Employee::getAge**”. A ternary operator is used to check for presence of **Optional** minimum value before retrieving it.
* As expected, the employee with minimum age is determined and printed as “**Nancy Smith Age:22 Salary:10000.0**”.

**Summary**  
In this tutorial we understood the working of the predefined **Collector** returned by the **Collectors.maxBy()** and **Collectors.maxBy()** methods. We looked at the formal definition of the **Collectors.maxBy()**/**Collectors.minBy()** methods, had a brief look at what actually happens inside the method, and finally saw the **Collectors.maxBy()**/**Collectors.minBy()** methods in action with a Java 8 code example followed by its explanation.

**Java 8 Joining with Collectors | Collectors.joining method tutorial with examples**

[March 13, 2017](https://www.javabrahman.com/java-8/java-8-joining-with-collectors-collectors-joining-method-tutorial-with-examples/)

Java 8 Joining with Collectors tutorial explains how to use the predefined **Collector** returned by **java.util.Stream.Collectors** class’ **joining()** method with examples. The ‘joining collector’ *collects*elements of a stream by concatenating them into a single **String.** **Collectors** class provided 3 overloaded **joining()** methods returning 3 variants of joining collectors. All variants return concatenated strings but with minor differences.

In this tutorial I will explain the three overloaded **joining()** methods starting with their formal definition, then explain their working, and finally shows the methods’ usage via Java code examples.  
*(Note – This tutorial assumes that you are familiar with basics of*[*Java 8 Collectors*](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/)*.)*

**Predefined Collector returned by Collectors.joining() method with no arguments**  
**Collectors.joining()** method is defined with the following signature –

**public static Collector<CharSequence, ?, String> joining()**

Where,  
     – output is a **Collector**, which collects a Stream of elements of type **CharSequence**, with its [finisher](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples#ref1)returning the ‘collected’ value of type **String**

The **Collector** returned by **Collectors.joining()** method returns a **String** which is the concatenation of all elements of the **Stream**. An important thing to keep in mind regarding the joining collector is that it needs a stream of type **java.lang.CharSequence** to be fed as input to it. In other words, joining collector can only concatenate streams whose elements are subclasses of **CharSequence** interface. **String**, **StringBuffer**, **StringBuilder** are some of the subclasses of **CharSequence** which you will be using in most of joining scenarios. It is also worthwhile to note that internally the joining collector uses a **StringBuilder** instance to concatenate the stream elements.

So, given a stream of type **String** with 3 values “a”, “b”, “c” and you intend to create a concatenated string with value “abc”,i.e. a straightforward concatenation of the stream elements, then **Collectors.joining()** method is what you need to pass to your stream’s [terminal operation](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/) **collect()**.

**What if your Stream elements aren't subclasses of CharSequence ?**

In that case you need to convert the stream elements to their corresponding **CharSequence** (subclass of **CharSequence** *rather*) equivalents using the **Stream.map()** method with a **Function<T,CharSequence>**instance. You can also use the **toString()** method in your Function definition to get the **String**equivalents of your stream elements, i.e. if the elements’ class has the required **toString()** format. Mapping to **toString()** will work because **String** class implements **CharSequence**.

**Java 8 code example showing Collectors.joining() method’s usage**

**Java 8 code example showing Collectors.joining() usage**

**package com.javabrahman.java8.collector;**

**import java.util.stream.Collectors;**

**import java.util.stream.Stream;**

**public class JoiningWithCollectors {**

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

**String joinedStr =**

**Stream.iterate(new Integer(0), (Integer integer) -> integer + 1)**

**.limit(5)**

**.map(integer -> integer.toString())**

**.collect(Collectors.joining());**

**System.out.println("Joined String: "+joinedStr);**

**}**

**}**

**OUTPUT of the above code**

Joined String:  01234

**Explanation of the code**

* In the **main()** method of **JoiningWithCollectors** class, first an [infinite stream](https://www.javabrahman.com/java-8/java-8-streams-api-creating-infinite-streams-with-iterate-and-generate-methods/) is created using **Stream.iterate()** method. The initial value of this **Stream** is set as **Integer(0)** and then each subsequent element is obtained by adding **1**.
* Infinite stream of integers is then limited to **5** elements using the [Stream.limit(5)](https://www.javabrahman.com/java-8/java-8-filtering-and-slicing-streams-tutorial-with-examples/) method.
* Next, **Stream** elements are mapped to their **String** equivalents using a [pipelined](https://www.javabrahman.com/programming-principles/pipelines-in-computing-and-software-engineering/) **Stream.map()** method which accepts as input a **Function<Integer,String>** which is specified using its lambda expression **integer -> integer.toString()**.
* Lastly, on this stream of Strings the **collect()** operation is invoked with **Collectors.joining()**returning the desired joining **Collector** instance.
* The final collected value returned from the **Stream** is **01234** which is then printed as output.

**Low readability of concatenated String returned by joining collector**  
If you notice carefully in the output of the above joining collector, then you will see that the elements which are concatenated together make one single *whole* string. It is difficult to find out which individual strings or characters were concatenated together to create the final *joined* string. Java designers noticed this aspect as well, and they have designed 2 overloaded **joining()** methods which make the concatenated string more comprehensible. The first of these methods allows inserting a delimiter between joined elements, while the second method puts a prefix and suffix for the entire joined string. Let us take a look at how these 2 overloaded **Collectors.joining()** methods work.

**Joining Collector which adds a delimiter between concatenated elements**  
Overloaded **Collectors.joining()** method which accepts a delimiter is defined with the following signature –

**public static Collector<CharSequence, ?, String> joining(CharSequence delimiter)**

Where,  
     – **delimiter** is the only input parameter of type **CharSequence**  
     – output is a **Collector**, acting on a Stream of elements of type **CharSequence**, with its [finisher](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples#ref1) returning the ‘collected’ value of type **String**

The **Collector** returned by **Collectors.joining()** method returns a **String** which is the concatenation of all elements of the **Stream**, with the delimiter passed as input inserted between concatenated elements.

So, given a stream of type **String** with 3 values “a”, “b”, “c”, and you intend to create a concatenated string with value “a,b,c”, then **Collectors.joining(",")** method is what you need to pass to your stream’s terminal operation **collect()**.

**Java 8 code example for Collectors.joining() method with a delimiter**  
*(Note – The class definition and imports are same as****Collectors.joining()****example above and hence are not repeated again for brevity)*

**Java 8 code example for Collectors.joining() with delimiter**

**package com.javabrahman.java8.collector;**

**import java.util.stream.Collectors;**

**import java.util.stream.Stream;**

**public class JoiningWithCollectors {**

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

**String joinedStr =**

**Stream.iterate(new Integer(0), (Integer integer) -> integer + 1)**

**.limit(5)**

**.map(integer -> integer.toString())**

**.collect(Collectors.joining(","));**

**System.out.println("Joined String: "+joinedStr);**

**}**

**}**

**OUTPUT of the above code**

Joined String:  0,1,2,3,4

**Explanation of the code**

* The code in the above example is exactly the same as that written for the plain **joining()** method that takes no input, except for the **collect()** operation where we pass the overloaded **joining()** method which takes a delimiter as input which in our case is a **“,”***(comma)*.
* The output is as expected – **0,1,2,3,4** i.e. final joined **String** with **“,”** inserted between elements as the delimiter.

The 3rd and last variant of joining collector is quite similar to the first 2 variants, except that it adds a suffix and prefix as well to the final joined string. Let us take a look at it now.

**Joining Collector which adds a delimiter between elements, and suffix & prefix to joined String**  
The signature of the 3rd variant of the joining collector is as follows –

**public static Collector<CharSequence, ?, String> joiningjoining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)**

Where,  
     – input param **delimiter** of type **CharSequence** is the delimiter which will be added between concatenated stream elements  
     – input param **prefix** of type **CharSequence** will be appended *before* the joined String  
     – input param **suffix** of type **CharSequence** will be appended *after* the joined String  
     – output is a **Collector**, acting on a Stream of elements of type **CharSequence**, with its finisher returning the ‘collected’ value of type **String**

The **Collector** returned by **Collectors.joining()** method returns a **String** which is the concatenation of all elements of the **Stream**, with the delimiter passed as input inserted between concatenated elements, and the prefix and suffix appended to the joined String.

So, given a stream of type **String** with 3 values “a”, “b”, “c”, and you intend to create a concatenated string with value “{a,b,c}”, then **Collectors.joining(",","{","}")** method is what you need to pass to your stream’s terminal operation **collect()**.

**Java 8 code example for Collectors.joining() method with a delimiter, prefix & suffix**  
*(Note – The class definition and imports are same as****Collectors.joining()****example above and hence are not repeated again for brevity)*

**Java 8 code example for Collectors.joining() with delimiter,prefix & suffix**

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

**String joinedStr =**

**Stream.iterate(new Integer(0), (Integer integer) -> integer + 1)**

**.limit(5)**

**.map(integer -> integer.toString())**

**.collect(Collectors.joining(",","{","}"));**

**System.out.println("Joined String: "+joinedStr);**

**}**

**OUTPUT of the above code**

Joined String:  {0,1,2,3,4}

**Explanation of the code**

* The code in the above example is exactly the same as that for the previous 2 overloaded variants of the **joining()** method, except for the **collect()** operation where we pass the 3rd variant of the **joining()**method which takes a delimiter, a prefix and a suffix as input parameters..
* The output is as expected – **{0,1,2,3,4}** i.e. a final *joined* **String** with **“,”** inserted between elements as the delimiter, and **“{”** & **“}”** as prefix and suffix respectively to the final joined **String**.

**Summary**  
In the above tutorial we understood how the *joining collector* returned by **java.util.Stream.Collectors.joining()** method works. We looked at the 3 overloaded variants of the **joining()** method and understood their working by going through their method signatures and Java 8 code examples showing three methods in use.

**Java 8 – Iterable.forEach, Iterator.remove methods tutorial with examples**

[October 21, 2016](https://www.javabrahman.com/java-8/java-8-iterable-foreach-iterator-remove-methods-tutorial-with-examples/)

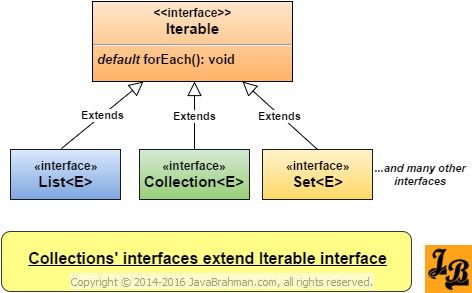
Collections API is the most popular and widely used utility API in the Java universe. With the advent of Functional Interfaces, Lambdas and Streams in Java 8, Collections API has also undergone changes to accomodate and build-upon the newly introduced functional programming features. The most widely used Collection classes and interfaces such as Iterable and Iterator, Collection, List and Map have all been enhanced in JDK 1.8 with new features and methods.

This is the **1st article in a 4 part article series** in which I will cover the **changes introduced in Java 8 Collections** in detail.

**In this part 1 of 4**, I will be covering the new default method named **forEach()** introduced in **java.lang.Iterable** interface with examples. This will be followed by understanding the new **default implementation of Iterator interface’s remove() method**, and how it makes implementing the Iterator easier than before.

**New default method forEach() added to the Iterable Interface in Java 8**

**Iterable** interface is a commonly extended interface among the Collections interfaces as it provides the ability to iterate over the members of a collection. **Collection**, **List** and **Set** are among the important Collections interfaces that extend **Iterable**, apart from other interfaces.



Java 8’s new **Iterable.forEach()** method has the following signature –

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

Where,  
     – **action** is the only parameter and is an instance of a [Consumer Functional Interface](https://www.javabrahman.com/java-8/java-8-java-util-function-consumer-tutorial-with-examples/)  
     – **forEach()** is implemented in the **Iterable** interface itself as a [default method](https://www.javabrahman.com/java-8/default-methods-in-java-8-with-examples/).

**Iterable.forEach()** method ‘consumes’ all the elements of the iterable collection of elements passed to it. The logic for consumption is passed to the method as an instance of a Consumer functional interface. An important point to note is that the **forEach method iterates *internally* over the collection of elements passed to it rather than *externally***. You can read about declarative internal iterators and how they differ from commonly used external iterators in this [tutorial here](https://www.javabrahman.com/java-8/java-8-internal-iterators-vs-external-iterators/).

Let us now take a look at Java code showing how to use **Iterable.forEach()** method for consuming the elements of an **Iterable** collection –

**Java 8 code to iterate and consume using Iterable.forEach() method**

**package com.javabrahman.java8.collections;**

**import java.util.Arrays;**

**import java.util.HashSet;**

**import java.util.List;**

**import java.util.Set;**

**public class IterableForEachExample {**

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

**List<Integer> intList= Arrays.asList(12,25,9);**

**System.out.println("List elements printed using Iterable.forEach");**

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

**Set<Integer> intSet=new HashSet<>();**

**intSet.add(50);**

**intSet.add(1);**

**System.out.println("Set elements printed using Iterable.forEach");**

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

**}**

**}**

**OUTPUT of the above code**

List elements printed using Iterable.forEach  
12  
25  
9  
Set elements printed using Iterable.forEach  
1  
50

**Explanation of the code**

* A **List** of primitive **int**s, named **intList**, is created using the **Arrays.asList()** method.
* **forEach()** method is invoked on the **List** instance – **intList**. As we read above, **List** inherits the **forEach()** method’s default implementation from **Iterable** interface.
* We pass a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/) to **System.out.println()** method, which is a **Consumer** type of function, as parameter to the **forEach()** method.
* **forEach()** method internally iterates and consumes, or prints, the elements of **intList**.
* Next we created an instance of a **Set**, named **intSet**, by using its concrete implementation **HashSet**.
* **Set** also inherits **Iterable**. Hence, we are able to print the elements in **intSet** using the **forEach()**method similar to the way we did with **intList**.

**New default method remove() added to Iterator interface in Java 8**  
**Prior to Java 8**, implementing **Iterator** interface without the support for **remove()** method implied that the designers had to override the method and throw an **UnsupportedOperationException**. Such an override was commonly done and over the years had become kind of staple in **Iterator** implementations not supporting the remove operation.

**With Java 8** arrived the feature of adding default implementations of methods in interfaces itself. Java designers have used this new feature and added a default implementation of the **remove()** method in the **Iterator** interface itself which throws **UnsupportedOperationException**.

**As a result**, **the practice of overriding the remove() method, whenever it wasn’t supported, has now been inverted to overriding the remove() method only when remove functionality has to be implemented**. This has **removed the unnecessary overhead** of overriding and throwing the **UnsuportedOperationException**everytime when implementing an **Iterator** without remove functionality.

The default implementation of **remove()** method in Java 8’s **Iterable** is as shown below –

**default void remove() {**

**throw new UnsupportedOperationException("remove");**

**}**

As you can see in the above code, the default implementation of **Iterator.remove()** method just throws an **UnsupportedOperationException** with message **"remove"**.

**Conclusion**  
In this tutorial we dipped our feet into the waters of Java 8 Collection Enhancements by taking a look at the changes in **Iterable** and **Iterator** interfaces.

**In the forthcoming parts** I will be taking you to the depths of Java 8 Collections enhancements, where I will explain the changes in important Collection classes viz. **Collection**, **List** and **Map** interfaces. Specifically, among the **Map** enhancements we will take a look at the new methods added in Java 8 which make ***multi-value maps*** handling easier than before.

**Java 8 – Collection.removeIf method tutorial with examples**

[October 25, 2016](https://www.javabrahman.com/java-8/java-8-collection-removeif-method-tutorial-with-examples/)

This is the **2nd article, of a 4 part article series, covering the important enhancements introduced in Collections API in Java 8**. The previous article, 1st part of the series([read here](https://www.javabrahman.com/java-8/java-8-iterable-foreach-iterator-remove-methods-tutorial-with-examples/)), explained the enhancements introduced in **Iterable** and **Iterator** interfaces in Java 8.

**In this part 2 of 4**, I will be explaining the new default method **removeIf()** which has been added to the **java.util.Collection** interface. To understand the intent with which this method has been introduced, we will start off with looking at how element removal from a Collection worked until Java 7. We will then look at the new **Collection.removeIf()** method introduced in Java 8 along with a working example showing its usage. Along with seeing the code for the two ways of achieving conditional element removal, we will also understand the performance improvement which Java 8’s new **removeIf()** method delivers as compared to the Java 7 style of accomplishing the same.

**Conditional removal from a Collection before Java 8**  
Until Java 7, iterating a Collection and removing elements from it based on a given condition involved the following steps –

1. Creating a for loop for iterating over the collection elements. The newer **for..each** loop could not be used here as it only allows reading the elements in a collection without the possibility of removing them. So, one will need to use the standard long-form for-loop.
2. To remove an element from a collection required access to its iterator. So, one had to use the Collection’s iterator to access the elements as well as handle the loop terminating condition.
3. Examine each element, as we iterate over the collection, and based on the given condition either remove the current element being examined or let it remain in the collection.

Let’s see a sample program showing how conditional removal from a Collection could be done till Java 7. The **Collection** we will use will be an **ArrayList**, and it will contain objects of type **Employee**.

**Java 7 code showing Collection handling using Iterator**

**//Employee.java(POJO class)**

**package com.javabrahman.java8;**

**public class Employee {**

**private String name;**

**private Integer age;**

**private Double salary;**

**public Employee(String name, Integer age, Double salary) {**

**this.name = name;**

**this.age = age;**

**this.salary = salary;**

**}**

**public String toString(){**

**return "Employee Name:"+this.name**

**+" Age:"+this.age**

**+" Salary:"+this.salary;**

**}**

**//getters and setters for name, age and salary go here**

**//standard equals() and hashcode() code go here**

**}**

**//CollectionRemoveIfExample.java**

**com.javabrahman.java8.collections;**

**import com.javabrahman.java8.Employee;**

**import java.util.ArrayList;**

**import java.util.Iterator;**

**import java.util.List;**

**public class CollectionRemoveIfExample {**

**static List<Employee> employeeList = new ArrayList<>();**

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

**employeeList.add(new Employee("Tom Jones", 45, 7000.00));**

**employeeList.add(new Employee("Harry Major", 25, 10000.00));**

**employeeList.add(new Employee("Ethan Hardy", 65, 8000.00));**

**employeeList.add(new Employee("Nancy Smith", 22, 12000.00));**

**employeeList.add(new Employee("Deborah Sprightly", 29, 9000.00));**

**for(Iterator empIterator=employeeList.iterator();**

**empIterator.hasNext();) {**

**Employee emp = empIterator.next();**

**if(emp.getAge() > 30){**

**empIterator.remove();**

**}**

**}**

**System.out.println("Employees below the age of 30");**

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

**}**

**}**

**OUTPUT of the above code**

Employees below the age of 30  
Employee Name: Harry Major   Age:25   Salary:10000.0  
Employee Name: Nancy Smith   Age:22   Salary:12000.0  
Employee Name: Deborah Sprightly   Age:29   Salary:9000.0

**Explanation of the code**

* **Employee** class has three main attributes – **name**, **age** and **salary**.
* An **ArrayList** of Employees is created, named **employeeList**, to which 5 **Employee** objects are added.
* Using iterator-based iteration in a for-loop the individual **Employee** objects in **employeeList** are visited. All employees aged above 30 years are removed using the **Iterator.remove()** method. The code for iteration and conditional removal of employee objects is in red color.
* The **employeeList** is printed post the conditional removal and, as expected, the output consists of 3 employees aged below 30 years.

**Time Complexity of removal using for loop and iterator in specific case of an ArrayList**  
We used a for-loop with an iterator to iterate over the **ArrayList**. A single loop implies a time complexity of **O(n)**.

Further, as an **ArrayList** stores elements in sequential storage in memory, the removal of an element from the middle implies that all the elements to the right of(or *after*) the removed element have to be moved 1 place each towards the left. This needs to be done in order to fill up the empty space left by the removed element. The movement of elements for these place would further require time proportional to **O(m)**.

Since we potentially will need to move the elements for every removal, the total complexity across all the removals equals **O(n) X O(m) ~ O(n2)**.

**Conditional Removal from a Collection using Java 8’s Collection.removeIf() default method**  
Java 8 has added support for functional programming features. One of the important in-built functional interface is **Predicate**. **Predicate**, or a condition checking function, checks the given input for a given condition and returns a boolean result for the same indicating whether the condition was met or not.  
(Note – If you haven’t worked with Java 8’s Predicate Functional Interface before then you can read the [tutorial on Predicate](https://www.javabrahman.com/java-8/java-8-java-util-function-predicate-tutorial-with-examples/).)

Java 8’s **java.util.Collection** interface has a new default method added to it named **removeIf()**. This method is defined with the following signature –

**default boolean removeIf(Predicate<? super E> filter)**

Where,  
     – **filter** is an instance of a Predicate Functional Interface.

**Collection.removeIf()** method works by applying the condition provided in the **Predicate** instance to all the elements in the Collection on which it is invoked. The elements which satisfy the condition are retained while the remaining are removed from the Collection.

Let us now see how the code for doing the same operation we did above, that of filtering out employees if their age > 30, would look using the new **Collection.removeIf** method.  
*(Note – I am leaving out the code for Employee class, along with package and import declarations below as they remain the same as Java 7 code above.)*

**Java 8 code showing Collection.removeIf() usage**

**public class CollectionRemoveIfExample {**

**static List<Employee> employeeList = new ArrayList<>();**

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

**employeeList.add(new Employee("Tom Jones", 45, 7000.00));**

**employeeList.add(new Employee("Harry Major", 25, 10000.00));**

**employeeList.add(new Employee("Ethan Hardy", 65, 8000.00));**

**employeeList.add(new Employee("Nancy Smith", 22, 12000.00));**

**employeeList.add(new Employee("Deborah Sprightly", 29, 9000.00));**

**employeeList.removeIf((Employee emp) -> emp.getAge() > = 30);**

**System.out.println("Employees below the age of 30");**

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

**}**

**}**

**OUTPUT of the above code**

Employees below the age of 30  
Employee Name: Harry Major   Age:25   Salary:10000.0  
Employee Name: Nancy Smith   Age:22   Salary:12000.0  
Employee Name: Deborah Sprightly   Age:29   Salary:9000.0

**Explanation of the code**

* The code above is same as the earlier iterator-based code except the single line in green color which has replaced the red colored code.
* **removeIf()** method is invoked on **employeeList** with the input **Predicate** being **(Employee emp) -> emp.getAge() > 30**.
* The input predicate, expressed using its equivalent [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/), specifies exactly the same condition used in iterator based code – to remove **Employee** objects from the employeeList if the employee’s age is greater than 30.
* **Employee** objects remaining in the **employeeList** post the **removeIf()** method invocation are then printed and, as expected, 3 employees are printed.
* Notice that **instead of 6 lines of iterator-based code of Java 7, we wrote only a single line Java 8 code using the Collection.removeIf()** method.

**Time Complexity of removal from an ArrayList using Collection.removeIf()**  
Collection traversal using an iterator was not designed only for removals. The programmer could do whatever they wanted with the Collection elements as they iterated over it. **Collection.removeIf()** method, on the other hand, is specifically meant for removals. Knowing the specificity of purpose of the **removeIf()** method, and the need to remove the overhead of shifting of ArrayList elements after a removal from the midde, Java designers overrode the default implementation of **removeIf()** in the **ArrayList** (possible as **ArrayList**implements **Collection**), and optimize the code while doing so to achieve a time complexity of **O(n)**.

**O(n2) Vs O(n) – Big performance improvement**  
Removal from an **ArrayList** using an iterator has time complexity of **O(n2)**, while the same operation when performed using Java 8’s new **Collection.removeIf()** method has a complexity of **O(n)**. This is a significant improvement in performance. If an application has large-sized ArrayLists then using the **Collection.removeIf()** method will result in the application being speeded up by an order of complexity, making the choice of the new method over the earlier one a no-brainer in such scenarios.

**Conclusion**  
In this **2nd part of the 4-part Java 8 Collection Enhancements Series**, we looked at how conditional removals from a **java.util.Collection** was done prior to Java 8. We then saw how Java 8’s new **Collection.removeIf()** method uses a **Predicate** instance to do the same conditional removal operation in significantly less time **[O(n) vs O(n2)]** and with concise, single-line code.

**In the remaining 2 articles of the series**, we will be looking at the important changes introduced in **List** and **Map** interfaces respectively. Specifically, the article on **List** interface will cover the newly added **sort()** and **replaceAll()** methods, while the article on **Map** interface will explain the new methods introduced in Java 8 which allow easier handling of ***multi-value maps***.

**Java 8 – List.sort, List.replaceAll methods tutorial with examples**

[October 27, 2016](https://www.javabrahman.com/java-8/java-8-list-sort-list-replaceall-methods-tutorial-with-examples/)

**This is the 3rd article, of a 4 part article series, covering the important enhancements which have been introduced in Collections API in Java 8**. While the 1st part of the series explained the enhancements introduced in **Iterable** and **Iterator** interfaces in Java 8([read 1st part](https://www.javabrahman.com/java-8/java-8-iterable-foreach-iterator-remove-methods-tutorial-with-examples/)), the 2nd part covered**Collection** interface’s new default method **removeIf()**([read 2nd part](https://www.javabrahman.com/java-8/java-8-collection-removeif-method-tutorial-with-examples/)).

**In this part 3 of 4**, I will be looking at **java.util.List** interface and will explain the new default methods which have been introduced in the interface in Java 8 viz. **List.sort()** and **List.replaceAll()**. We will understand the working of these two methods along with seeing examples showing their usage.

**New default method replaceAll() added to the List Interface in Java 8**  
The new default method **List.replaceAll()** has been written for scenarios where a particular action has to be performed on *all* the elements of a list. **List.replaceAll()** is thus a *bulk mutation operation*.

An important point to note regarding **replaceAll()** method is that it does NOT change the type of the elements in the list. So, a list of say Strings will continue to contain Strings but using **replaceAll()** you can apply some common function on all the Strings in the list. An example of a **replaceAll()** call on a list of Strings would be to CAPITALIZE all Strings and the like.

Let us now take a look at the signature of the **List.replaceAll()** method –

**default void replaceAll(UnaryOperator<E> operator)**

Where,  
     – **operator** is the only parameter and is an instance of a **UnaryOperator** Functional Interface.

**What is a UnaryOperator**

**java.util.function.UnaryOperator** is a functional interface, and is a specialization of [Function](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/) with the operand and the return value being of the same type. I.e. **UnaryOperator<E>** takes a single input of type **E**, and returns an output of the same type **E**.

Let us now see an example of using **List.replaceAll()** to do a specific action on all elements of a **List**. In the below code snippet, each of the 5 employees of some company are being given a salary hike of 10% across the board using **List.replaceAll()** method.

**Java 8 code to do a specific change on all elements of a List using List.replaceAll()**

**package com.javabrahman.java8;**

**public class Employee {**

**private String name;**

**private Integer age;**

**private Double salary;**

**public Employee(String name, Integer age, Double salary) {**

**this.name = name;**

**this.age = age;**

**this.salary = salary;**

**}**

**public String toString(){**

**DecimalFormat dformat = new DecimalFormat(".##");**

**return "Employee Name:"+this.name**

**+" Age:"+this.age**

**+" Salary:"+dformat.format(this.salary);**

**}**

**//getters and setters for name, age and salary go here**

**//standard equals() and hashcode() code go here**

**}**

**//ListReplaceAllExample.java**

**package com.javabrahman.java8.collections;**

**import com.javabrahman.java8.Employee;**

**import java.util.Arrays;**

**import java.util.List;**

**public class ListReplaceAllExample {**

**static List<Employee> employeeList = Arrays.asList(**

**new Employee("Tom Jones", 45, 7000.00),**

**new Employee("Harry Major", 25, 10000.00),**

**new Employee("Ethan Hardy", 65, 8000.00),**

**new Employee("Nancy Smith", 22, 12000.00),**

**new Employee("Deborah Sprightly", 29, 9000.00));**

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

**employeeList.replaceAll(employee -> {**

**employee.setSalary(employee.getSalary() \* 1.1);**

**return employee;**

**});**

**System.out.println("Employee list with all salaries incremented by 10%");**

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

**}**

**}**

**OUTPUT of the above code**

Employee list with all salaries incremented by 10%  
Employee Name: Tom Jones   Age:45   Salary:7700.0  
Employee Name: Harry Major   Age:25   Salary:11000.0  
Employee Name: Ethan Hardy   Age:65   Salary:8800.0  
Employee Name: Nancy Smith   Age:22   Salary:13200.0  
Employee Name: Deborah Sprightly   Age:29   Salary:9900.0

**Explanation of the code**

* **Employee** class has 3 main attributes – **name**, **age** and **salary**.
* We create a List of **Employee** objects, named **employeeList**, and add 5 **Employee** objects to it.
* **replaceAll()** method is invoked on the **employeeList** with the input being a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/)equivalent to a **UnaryOperator** function – **(employee -> {employee.setSalary(employee.getSalary() \* 1.1);return employee;});**
* This lambda expression takes an **employee** as input and sets the salary of the employee at a value which is 10% more than the employee’s current salary.
* Lastly, the **List** of employees is printed. The salaries of all the employees has been increased by 10% as expected.

**New default method sort() added to the List Interface in Java 8**  
**List.sort()** is a new default method introduced in Java 8 which sorts the given list based on the **Comparator**instance passed to it as input. Let us start understanding the method by first looking at its signature –

**default void sort(Comparator<? super E> c)**

Where,  
     – **c** is the only parameter and is an instance of a [Comparator](https://www.javabrahman.com/java-8/the-complete-java-8-comparator-tutorial-with-examples/) which is an ancestor of type **E**, where **E** is the type of the elements in the **List** being sorted.

**Why create a new method List.sort() when Collections.sort() was already there**  
Prior to Java 8, **Collections.sort()** method was popularly used to lists. However, there was a drawback with **Collections.sort()** that it doesn’t sort in-place. An in-place sort saves both on memory (by not requiring space other than that occupied by its elements) and time ( as the twin tasks of creating a temporary copy of the list to be sorted and then copying the sorted elements back into the original list are no longer required).

**List.sort()**, however, does use an in-place variant of merge sort to sort the **List** elements. As a result it provides both the space and time benefits mentioned above. In fact, in Java 8, the **Collections.sort()**method itself internally calls **List.sort()** to sort the **List** elements.

Let us now take a look an example showing **List.sort()** usage. We will use the same list of employees as we used in the previous code snippet and sort them in the order of increasing order of their salary.  
*(Note – I am leaving out the code for****Employee****class below as it is the same as used in the****replaceAll()****code above.)*

**Java 8 code to sort a List using List.sort() default method**

**package com.javabrahman.java8.collections;**

**import com.javabrahman.java8.Employee;**

**import java.util.Arrays;**

**import java.util.List;**

**public class ListSortExample {**

**static List<Employee> employeeList = Arrays.asList(**

**new Employee("Tom Jones", 45, 7000.00),**

**new Employee("Harry Major", 25, 10000.00),**

**new Employee("Ethan Hardy", 65, 8000.00),**

**new Employee("Nancy Smith", 22, 12000.00),**

**new Employee("Deborah Sprightly", 29, 9000.00));**

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

**employeeList.sort((emp1, emp2)->**

**Double.compare(emp1.getSalary(),emp2.getSalary()));**

**System.out.println("Employee list sorted by their salaries");**

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

**}**

**}**

**OUTPUT of the above code**

Employee list sorted by their salaries  
Employee Name: Tom Jones   Age:45   Salary:7000.0  
Employee Name: Ethan Hardy   Age:65   Salary:8000.0  
Employee Name: Deborah Sprightly   Age:29   Salary:9000.0  
Employee Name: Harry Major   Age:25   Salary:10000.0  
Employee Name: Nancy Smith   Age:22   Salary:12000.0

**Explanation of the code**

* A list of 5 **Employee** objects is created, named **employeeList**.
* **sort()** method is invoked on **employeeList** with the lambda expression equivalent of a **DoubleComparator** being passed to it – **(emp1, emp2)-> Double.compare( emp1.getSalary(),emp2.getSalary() )** .
* Definition of **Double.compare()** method matches the [function descriptor](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/) of **Comparator** interface, allowing us to use the above lambda expression.
* The sorted **employeeList**‘s **Employee** objects are then printed. As expected the **Employee** objects have been sorted in the increasing order of their salary.

**Conclusion**  
In this **3rd part of the 4-part Java 8 Collection Enhancements Series**, we looked at the new default methods **List.replaceAll()** and **List.sort()** introduced in Java 8. We understood the working of these new methods and also saw examples showing their usage.

**In the next(and ultimate) article of the series**, we will be looking at the Java 8’s new Collection API enhancements introduced in **Map** interface which ease the effort required to handle **multi-value maps** *aka***multimaps**.

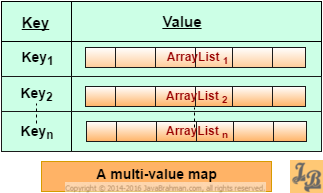
**Java 8 – Map’s computeIfAbsent, computeIfPresent, getOrDefault methods tutorial with examples**

[November 4, 2016](https://www.javabrahman.com/java-8/java-8-maps-computeifabsent-computeifpresent-getordefault-methods-tutorial-with-examples/)

**This is the last article, of a 4 part article series, covering the significant enhancements in Java 8 Collections API. While the 1st part of the series explained the new default methods introduced inIterable and Iterator interfaces in Java 8(**[read 1st part](https://www.javabrahman.com/java-8/java-8-iterable-foreach-iterator-remove-methods-tutorial-with-examples/)**), the 2nd part covered the new default method removeIf() introduced in Collection interface, (**[read 2nd part](https://www.javabrahman.com/java-8/java-8-collection-removeif-method-tutorial-with-examples/)**), and the 3rd explained Listinterface’s new sort() and replaceAll() methods(**[read 3rd part](https://www.javabrahman.com/java-8/java-8-maps-computeifabsent-computeifpresent-getordefault-methods-tutorial-with-examples/www.javabrahman.com/java-8/java-8-list-sort-list-replaceall-methods-tutorial-with-examples/)**).**

**In this tutorial**, we will be looking at the enhancements introduced in **java.util.Map** interface in Java 8. We will first quickly understand what are multi-value maps. Next we will create a multi-value map which will serve as the base problem set for explaining the new Map methods. We will first see the working of Java 8’s new **Map.forEach()** and **Map.replaceAll()** methods. Next, we will understand the new default methods introduced in Java 8 which simplify using multi-value maps. These methods are **Map.computeIfAbsent()**, **Map.computeIfPresent()** and **Map.getOrDefault()** methods.

**What is a multi-value map**  
A multi-value map is a normal instance of **java.util.Map**. The only difference is that instead of having a ‘single’ value corresponding to each key, a multi-value map instead has a ‘collection’ such as a **List** or a **Set** as the value stored against each key.



**A good example of a use-case for a multi-value map would be of the hash table** used to store hash keys and corresponding values. In a hash table, there exists a mappping between a hash value(key) and entries stored in a bucket (value) stored corresponding to it. A hash table is thus essentially a multi-value map.

Having understood the structure of a multi-value map, let us now define the base data set for showing the working of the new **Map** methods –

**Defining the data set/multi-value Map for this tutorial**

**Java 8 code defining a multi-value map**

|  |
| --- |
| package com.javabrahman.java8;  public class Employee {    private String name;    private Integer age;    private Double salary;    public Employee(String name, Integer age, Double salary) {      this.name = name;      this.age = age;      this.salary = salary;    }    public String toString(){      DecimalFormat dformat = new DecimalFormat(".##");      return "Employee Name:"+this.name          +"  Age:"+this.age          +"  Salary:"+dformat.format(this.salary);    }  //getters and setters for name, age and salary go here  //standard equals() and hashcode() code go here  }  //MultiValueMapsExample.java  package com.javabrahman.java8.collections;  import com.javabrahman.java8.Employee;  import java.util.\*;  public class MultiValueMapsExample {    static Map<Integer, List<Employee>> employeeDOJMap = new HashMap<>();      public static void main(String args[]) {        List<Employee> list2014 = Arrays.asList(          new Employee("Deborah Sprightly", 29, 9000.00));      employeeDOJMap.put(2014, list2014);      List<Employee> list2015 = Arrays.asList(          new Employee("Tom Jones", 45, 7000.00),          new Employee("Harry Major", 25, 10000.00));      employeeDOJMap.put(2015, list2015);      List<Employee> list2016 = Arrays.asList(          new Employee("Ethan Hardy", 65, 8000.00),          new Employee("Nancy Smith", 22, 12000.00));      employeeDOJMap.put(2016, list2016);    }  } |

**Explanation of the code**

* **Employee** class is the POJO for employees. It contains three attributes of an employee – **name**, **age** and **salary**.
* **employeeDOJMap** is the multi-value map of type **Map<Integer, List<Employee>>**.
* **employeeDOJMap**’s **Integer** key contains the year of joining of employees as the key, while the **List<Employee>>** stored as value for each key contains the instances of **Employee** objects who joined in that year.

***Important Note – In the next sections of this tutorial we will be using the base data set defined in the multi-value map named employeeDOJMap. Only the delta/new code in further sections for brevity.***

**Java 8’s new Map.forEach() and Map.replaceAll() methods**  
Let us start by quickly going through the relatively simpler methods introduced in Map interface in Java 8- **Map.forEach()** and **Map.replaceAll()**.

**Map.forEach()**  
**Map.forEach()** method is defined as –

**default void forEach(BiConsumer<? super K, ? super V> action)**

Where,  
     – **action** is the only parameter and is an instance of a BiConsumer functional interface, and,   
     – method applies the logic provided via **action** to all the entries in the map as it ‘consumes’ them.

**What is a BiConsumer**

**java.util.function.BiConsumer** is a functional interface, and is a two-arity specialization of a [Consumer Functional Interface](https://www.javabrahman.com/java-8/java-8-java-util-function-consumer-tutorial-with-examples/). I.e. it accepts two inputs as arguments and does not return any output.

**Map.replaceAll()**  
**Map.replaceAll()** method is defined as –

**default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)**

Where,  
     – **function** is the only parameter and is an instance of a **BiFunction** Functional Interface, and,   
     – the method applies the logic provided via **function** to all the values in the map and transforms them inside the map itself.

**What is a BiFunction**

**java.util.function.BiFunction** is a functional interface, and is a two-arity specialization of [Function](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/). I.e. it accepts two inputs as arguments and returns a result after performing a computation with the input.

Let us now see the **Map.forEach()** and **Map.replaceAll()** methods in action –

**Java 8 code showing Map.forEach() and Map.replaceAll() methods**

|  |
| --- |
| System.out.println("Using Map.forEach to print the Employee in employeeDOJMap multi-value map\n");  employeeDOJMap.forEach((year,empList)->System.out.println(year+"-->" +empList));    System.out.println("\nCAPITALIZED Employee Names using Map.replaceAll()");  employeeDOJMap.replaceAll((year, empList) -> {  empList.replaceAll(emp -> {      emp.setName(emp.getName().toUpperCase());      return emp;    });    return empList;  });  employeeDOJMap.forEach((year, empList)-> System.out.println(year+"-->"+empList)); |

**OUTPUT of the above code**

Using Map.forEach to print the Employee in employeeDOJMap multi-value map –

2016–>[Employee Name: Ethan Hardy  Age: 65  Salary: 8000.0, Employee Name: Nancy Smith  Age: 22  Salary: 12000.0]

2014–>[Employee Name: Deborah Sprightly  Age: 29  Salary: 9000.0]

2015–>[Employee Name: Tom Jones  Age: 45  Salary: 7000.0, Employee Name: Harry Major  Age:25  Salary: 10000.0]

CAPITALIZED Employee Names using Map.replaceAll()

2016–>[Employee Name: ETHAN HARDY  Age: 65  Salary: 8000.0, Employee Name: Nancy Smith  Age: 22  Salary: 12000.0]

2014–>[Employee Name: DEBORAH SPRIGHTLY  Age: 29  Salary: 9000.0]

2015–>[Employee Name: TOM JONES  Age: 45  Salary: 7000.0, Employee Name: Harry Major  Age:25  Salary: 10000.0]

**Explanation of the code**

* **Map.forEach()** method is passed the [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) equivalent to a **BiConsumer** implementation which prints the keys(years) and the corresponding values(employee lists) stored in the **employeeDOJMap**.
* **Map.replaceAll()** method uses a lambda equivalent of a **BiFunction** implementation. This lambda uses a **List.replaceAll()** method to replace all the employee names stored inside each of the **List<Employee>**, with their capitalized versions.

**Java 8’s Map.computeIfAbsent() method**  
As we learnt earlier, a multi-value map stores a collection of values for each key. Lets say we are adding a [key,value] entry to a multi-value map and the key we are adding is not present in the map. This would require for us to check for this probability before insertion and if the key is not present then we will have to instantiate a fresh collection instance as the value for this new key. Only then can we store the value against the key. Also, this check will need to be performed on every insert that we do.

**Map.computeIfAbsent()** takes away exactly this overhead of writing the multiple line code for such a check by squeezing it into a simple one line code. **Map.computeIfAbsent()** method is defined as –

**default V computeIfAbsent(K key,Function<? super K,? extends V>mappingFunction)**

Where,  
     – **key** is the first parameter which is the key of the multi-value map.   
     – **function** is an instance of **java.util.function.Function**. It computes and returns the value which is to be used when the key is new i.e. does not have a collection instantiated in the case of a multi-value map.

Have a look at the code sample below to understand the difference in code when using Java 7 versus Java 8’s **computeIfAbsent()** method. The red code is Java 7 way of checking and instantiating before all insertions. The green code is Java 8 way of doing the same thing using **Map.computeIfAbsent()** method.

**Java 8 code to show usage of Map.computeIfAbsent() method**

|  |
| --- |
| System.out.println("\nJava 7 way of adding a new key(2017) in a multi-value map\n"); |

**List empList2017 = employeeDOJMap.get(2017);**

**if (empList2017 == null) {**

**empList2017 = new ArrayList<>();**

**}**

**empList2017.add(new Employee("Tom Newman", 45, 12000.00));**

**employeeDOJMap.put(2017, empList2017);**

|  |
| --- |
| employeeDOJMap.forEach((year,empList)-> System.out.println(year+"-->"+empList));  System.out.println("\nUsing Map.computeIfAbsent() to add a new key(2018) in a multi-value map\n"); |

**employeeDOJMap.computeIfAbsent(2018,empList -> new ArrayList<>())**

**.add(new Employee("Dick Newman", 35, 10000.00));**

|  |
| --- |
| employeeDOJMap.forEach((year,empList)-> System.out.println(year+"-->"+empList)); |

**OUTPUT of the above code**

Java 7 way of adding a new key(2017) in a multi-value map

2016–>[Employee Name: Ethan Hardy  Age: 65  Salary: 8000.0, Employee Name: Nancy Smith  Age: 22  Salary: 12000.0]

2017–>[Employee Name: Tom Newman  Age: 45  Salary: 12000.0]

2014–>[Employee Name: Deborah Sprightly  Age: 29  Salary: 9000.0]

2015–>[Employee Name: Tom Jones  Age: 45  Salary: 7000.0, Employee Name: Harry Major  Age:25  Salary: 10000.0]

Using Map.computeIfAbsent() to add a new key(2018) in a multi-value map

2016–>[Employee Name: Ethan Hardy  Age: 65  Salary: 8000.0, Employee Name: Nancy Smith  Age: 22  Salary: 12000.0]

2017–>[Employee Name: Tom Newman  Age: 45  Salary: 12000.0]

2018–>[Employee Name: Dick Newman  Age: 35  Salary: 10000.0]

2014–>[Employee Name: Deborah Sprightly  Age: 29  Salary: 9000.0]

2015–>[Employee Name: Tom Jones  Age: 45  Salary: 7000.0, Employee Name: Harry Major  Age:25  Salary: 10000.0]

**Explanation of the code**

* The red code shows the Java 7 way of adding an employee named **Tom Newman** for the key **2017**. **It is a 5-6 line code.**
* The green code shows Java 8’s **computeIfAbsent()** method usage. A lambda expression implementing the **BiFunction** logic is passed to the method which instantiates and an **ArrayList** instance if it is empty(i.e. absent). To this **ArrayList** instance is added the new employee named **Dick Newman**. **It is a one line code.**
* The output shows the **employeeDOJMap** printed using **map.forEach()** method after each of the two additions. This is shown to see that **there is no difference in the way the new entries(key-value pairs) in the multi-value map are stored when using the Java 7 way and the Java 8 computeIfAbsent() method.**

**Java 8’s new Map.computeIfPresent() method**  
When working with multi-value maps, there are scenarios when while deleting a [key,value] pair, we might be removing the last/only value in the collection stored as the value for that key. In such cases, after removing the value from the collection, we would want to release the memory occupied by that the empty collection by removing that key from the multi-value map itself. This would require us to check after ‘every’ removal whether the value being removed is the last value for that key, and if so, remove the key from the multi-value map.

**Map.computeIfPresent()** takes away exactly this overhead of checking after every removal by reducing it to a simple one-line code. **Map.computeIfPresent()** method is defined as –

**default V computeIfPresent(K key,BiFunction<? super K,? super V,? extends V>remappingFunction)**

Where,  
     – **key** is the first parameter which is the key of the multi-value map.   
     – **remappingFunction** is an instance of a **BiFunction**. It computes and returns a value. In case of multi-value maps, the outcome of the key’s collection is decided based the value returned by this function. I.e. whether to keep the collection(if the collection is returned) or delete the collection(if a null value is returned).

To understand the usage of **computeIfPresent()** method better, let us have a look at the code sample below to understand the difference in code when using Java 7 versus Java 8’s **computeIfPresent()** method. As we did for the previous method, we will use color coding to differentiate between the Java 7 and Java 8 code. The red code is Java 7 way of checking and removing a key after every removal from the map. The green code is Java 8 way of doing the same thing using **Map.computeIfPresent()** method.

**Java 8 code to show usage of Map.computeIfPresent() method**

|  |
| --- |
| System.out.println("\nJava 7 way of removing a key(2017) in a multi-value map for which no entry exists\n"); |

**List empListDel = employeeDOJMap.get(2017);**

**empListDel.removeIf(employee -> employee.getName().equals("Tom Newman"));**

**if (empListDel.size() == 0) {**

**employeeDOJMap.remove(2017);**

**}**

|  |
| --- |
| employeeDOJMap.forEach((year, empList)-> System.out.println(year+"-->"+empList));  System.out.println("\nUsing Map.computeIfPresent() to remove a key(2018) for which no entry exists\n"); |

**employeeDOJMap.computeIfPresent(2018, (year, empList) -> empList.removeIf(employee -> employee.getName().equals("Dick Newman")) && empList.size() == 0 ? null : empList);**

|  |
| --- |
| employeeDOJMap.forEach((year, empList)-> System.out.println(year+"-->"+empList)); |

**OUTPUT of the above code**

Java 7 way of removing a key(2017) in a multi-value map for which no entry exists

2016–>[Employee Name: Ethan Hardy  Age: 65  Salary: 8000.0, Employee Name: Nancy Smith  Age: 22  Salary: 12000.0]

2018–>[Employee Name: Dick Newman  Age: 35  Salary: 10000.0]

2014–>[Employee Name: Deborah Sprightly  Age: 29  Salary: 9000.0]

2015–>[Employee Name: Tom Jones  Age: 45  Salary: 7000.0, Employee Name: Harry Major  Age:25  Salary: 10000.0]

Using Map.computeIfPresent() to remove a key(2018) for which no entry exists

2016–>[Employee Name: Ethan Hardy  Age: 65  Salary: 8000.0, Employee Name: Nancy Smith  Age: 22  Salary: 12000.0]

2014–>[Employee Name: Deborah Sprightly  Age: 29  Salary: 9000.0]

2015–>[Employee Name: Tom Jones  Age: 45  Salary: 7000.0, Employee Name: Harry Major  Age:25  Salary: 10000.0]

**Explanation of the code**

* The red code shows the Java 7 way of removing the newly added employee for **2017** named **Tim Newman**. **List.removeIf()** method is used for finding out the employee and removing him. **It is a 4-5 line code.**
* The green code shows Java 8 way of removing the only employee stored in the **List** for year **2018** – **Dick Newman**. It uses the **computeIfPresent()** method, to which a lambda expression equivalent to the BiFunction implementation is passed. This lambda checks if the list is empty after removing the aforesaid employee. If the list is empty the lambda returns a **null** else it returned the **empList** object itself. The **computeIfPresent()** method deletes the entry for the key 2018 if null is returned. **This is a one line code, *abeit a long line of code*.**
* The output shows the **employeeDOJMap** printed using **map.forEach()** method after each of the two additions. This is shown to see that **there is no difference in the way entries(key-value pairs) are removed from the multi-value map when using the Java 7 way as compared to the Java 8 computeIfAbsent()way.**

**Map.getOrDefault() method**  
**Map.getOrDefault()** method has been designed for scenarios where the value returned for a given key might be null i.e. the given key is not present in the map. In case of multi-value maps, it gives the programmer a utility to avoid **NullPointerException** at runtime by instantiating a new collection instance and returning it in case the key is not present and a null-value would otherwise have been returned.

**Map.getOrDefault()** is defined as follows –

**default V getOrDefault(Object key, V defaultValue)**

Where,  
     – **key** is the first parameter which is the key of the multi-value map.   
     – **defaultValue** is the value which will be used as default in case the key is not present and a null is returned.

To understand the working of the **Map.getOrDefault()** method better letter look at the code snippet showing its usage-

**Java 8 code to show usage of Map.getOrDefault() method**

|  |
| --- |
| System.out.println("\nAvoiding a null return when fetching a non-existent key's entry using Map.getOrDefault() method\n");  List<Employee> empList2019 = employeeDOJMap.getOrDefault(2019, new ArrayList<>());  System.out.println("Size of empList 2019 = " + empList2019.size()); |

**OUTPUT of the above code**

Avoiding a null return when fetching a non-existent key’s entry using Map.getOrDefault() method

Size of empList 2019 = 0

**Explanation of the code**

* **getOrDefault()** method is used to fetch the value for the key **2019**. The default value is set as a new empty **ArrayList()**.
* Since the key **2019** does not exist, the empty **ArrayList** is returned instead. The size of this **ArrayList** is then printed. As expected the value of size is printed as **0**.

**Conclusion**  
In this tutorial we started by understanding the concept of multi-value maps. We then defined the multi-value map we used throughout the tutorial as we understood the definition and usage of the new methods added in **Map** interface in Java 8. The new **Map** methods we covered in detail are – **forEach()**, **replaceAll()**, **computeIfAbsent()**, **computeIfPresent()** and **getOrDefault()** method. With this we conclude the 4-part series covering the Java 8 Collection enhancements.

**The Complete Java 8 Comparator Tutorial with examples**

[September 2, 2016](https://www.javabrahman.com/java-8/the-complete-java-8-comparator-tutorial-with-examples/)

**Do You Know – Comparator in Java 7 had just 2 methods – compare() and equals(). The enhanced Comparator in Java 8 now boasts of 19 methods. Yes, 17 more methods! What’s more Comparator now is an official Functional Interface as well!**

**Comparator has undergone a major overhaul in Java 8** while still retaining its essence which is to compare and sort objects in Collections. Comparator now supports **declarations via**[lambda expressions](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/)**as it is a Functional Interface**. Comparator has a **new method comparing()** which uses an instance of**java.util.function.Function** functional interface, specified using lambda expression or its equivalent method reference, for **Comparator** instance creation. In addition, **multiple sort criteria can now be clubbed using comparing() with a thenComparing()** method. The range of new capabilities is rounded off with methods for using **natural comparison order, in-built Null handling and sort order reversal**.

In this tutorial, we will first take a quick look at how Comparators were used before Java 8. We will then take an in-depth look at the new Comparator aspects mentioned above to see how **java.util.Comparator** has evolved into an enhanced comparison and ordering utility in Java 8.

**How Comparator was used prior to Java 8**  
Until Java 7, **Comparator** interface could only be used in one single way. Given a collection of objects of type **<T>** to sort, one would create an implementation of **Comparator<T>** interface, override the **compare()**method of the interface with the desired comparison logic and use **Collections.sort()** or similar such methods in Collections API to sort the collection of objects.

Let us now see an example of how Comparators were used prior to Java 8. Let us first create a Class **Employee**which will be the type of object we will be sorting across all our examples for Java 7 and Java 8 Comparators-

**Employee.java**

|  |
| --- |
| //Employee.java  package com.javabrahman.java8;  public class Employee{    private String name;    private Integer age;    public Employee(String name, Integer age){      this.name=name;      this.age=age;    }    //--getters/setters for name and age go here    public String toString(){      return "Employee Name:"+this.name        +"  Age:"+this.age;    }    @Override    public boolean equals(Object obj) {       if (obj == this) {         return true;       }       if (!(obj instanceof Employee)) {         return false;       }       Employee empObj = (Employee) obj;         return this.age==empObj.age           && this.name.equalsIgnoreCase(empObj.name);    }    @Override    public int hashCode() {      int hash = 1;      hash = hash \* 17 + this.name.hashCode();      hash = hash \* 31 + this.age;      return hash;    }  } |

Next we will see how Comparator was used to sort collections prior to Java 8 –

**How Comparator was used prior to Java 8**

|  |
| --- |
| package com.javabrahman.java8.comparator;  import java.util.Comparator;  import com.javabrahman.java8.Employee;  public class EmployeeComparator implements Comparator<Employee> {    @Override    public int compare(Employee emp1, Employee emp2) {      return (emp1.getName().compareTo(emp2.getName()));    }  }  //ComparatorOldWay.java  package com.javabrahman.java8.comparator;  import java.util.Arrays;  import java.util.Collections;  import java.util.List;  import com.javabrahman.java8.Employee;  public class ComparatorOldWay {    static List<Employee> employeeList =        Arrays.asList(new Employee("Tom Jones", 45),          new Employee("Harry Major", 35),          new Employee("Harry Major", 25),          new Employee("Ethan Hardy", 65),          new Employee("Nancy Smith", 15),          new Employee("Deborah Sprightly", 29));      public static void main(String args[]) {      Collections.sort(employeeList, new EmployeeComparator());      employeeList.forEach(System.out::println);    }  } |

**OUTPUT of the above code**

Employee Name:Deborah Sprightly Age:29  
Employee Name:Ethan Hardy Age:65  
Employee Name:Harry Major Age:35  
Employee Name:Harry Major Age:25  
Employee Name:Nancy Smith Age:15  
Employee Name:Tom Jones Age:45

**Quick Explanation of the above code-**

* **EmployeeComparator** class implements **Comparator** inteface and overrides the **compare()** method to compare **Employee** objects passed to it based on the *natural* ordering of their names of **String** type.
* **ComparatorOldWay** class sorts a static list of **Employee** objects using an instance of **EmployeeComparator** and the **Collections.sort()** method.
* Output shows that the **employeeList** gets sorted alphabetically based on the names of the employees.

**Java 8’s Comparator is a Functional Interface**  
Owing to the fact that the Comparator interface has just one abstract method, **compare()**, it automatically qualifies to be a [Functional Interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) in Java 8. Nevertheless, Java 8 designers have gone ahead and annotated the **Comparator** class interface **@FunctionalInterface** to enforce its role as a Functional Interface. Being a functional interface, Comparator can now be used as an assignment target for a lambda expression or a method reference.

**Java 8’s Comparator as an assignment target for LambdaExpressions**  
Given the fact that its a functional interface, an instance of a Comparator can now be created in Java 8 with a lambda expression specifying its comparison logic. Take a look at the code snippet below –

**Defining a Comparator using lambda expression**

|  |
| --- |
| package com.javabrahman.java8.comparator;  import java.util.Arrays;  import java.util.Collections;  import java.util.Comparator;  import java.util.List;  import com.javabrahman.java8.Employee;  public class ComparatorsInJava8 {    static List<Employee> employeeList =        Arrays.asList(new Employee("Tom Jones", 45),          new Employee("Harry Major", 35),          new Employee("Harry Major", 25),          new Employee("Ethan Hardy", 65),          new Employee("Nancy Smith", 15),          new Employee("Deborah Sprightly", 29));    public static void main(String[] args) {      Comparator<Employee> empNameComparator = (Employee emp1, Employee emp2) -> {        return (emp1.getName().compareTo(emp2.getName()));      };      Collections.sort(employeeList, empNameComparator);      employeeList.forEach(System.out::println);    }  } |

**OUTPUT of the above code**

Employee Name:Deborah Sprightly Age:29  
Employee Name:Ethan Hardy Age:65  
Employee Name:Harry Major Age:35  
Employee Name:Harry Major Age:25  
Employee Name:Nancy Smith Age:15  
Employee Name:Tom Jones Age:45

**Quick Explanation of the above code-**

* **ComparatorsInJava8** class uses the same list of **Employee** objects as that used for the previous example for Java 7 style of **Comparator**.
* An instance of the **Comparator**, **empNameComparator**, is created using a lambda expression.
* The lambda expression takes 2 **Employee** instances,**emp1** and **emp2**, as input and outputs the comparison of their names using the natural comparison order of Strings.
* Using **empNameComparator** for sorting results in a correctly sorted **Employee** list by name.

**Java 8 Comparator’s comparing() method’s working**  
The **comparing()** method is a new static method introduced in Comparators in Java 8. It has the signature –

**static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor)**

**comparing()** method works by taking a **Function<T,R>** functional interface instance as an input, where **T** is the type of input object and **R** is the sort key which is returned (or extracted) from the input object when **Function<T,R>** processes it.

Let’s see a code snippet to understand the use of **comparing()** method. I have removed the code which was same as previous example below for brevity –

**Comparator creation using static method comparing()**

|  |
| --- |
| public static void main(String[] args) {   Comparator<Employee> comparatorObj=Comparator.comparing(Employee emp -> emp.getName());   Collections.sort(employeeList, comparatorObj);   employeeList.forEach(System.out::println);  } |

**OUTPUT of the above code**

Employee Name:Deborah Sprightly Age:29  
Employee Name:Ethan Hardy Age:65  
Employee Name:Harry Major Age:35  
Employee Name:Harry Major Age:25  
Employee Name:Nancy Smith Age:15  
Employee Name:Tom Jones Age:45

**Quick Explanation of the above code-**

* An instance of the **Comparator**, **comparatorObj**, is created using the **static** method **Comparator.comparing()**.
* The **comparing()** is passed a lambda expression, which corresponds to a **Function<T,R>** instance accepting an **Employee** object as input and returns an employee name – *the sort key*.
* Using **comparatorObj** for sorting results in a correctly sorted **Employee** list by name.
* **NOTE** – Instead of the lambda expression, you can also use an equivalent method reference as well. The **comparing() method with a method reference** will then be written like this –

**Comparator empNameComparator = Comparator.comparing(Employee::getName);**

**Java 8 Comparator’s thenComparing() method for multiple sort criteria**  
Many-a-times we need to sort with multiple sort orders. I.e. on more than one attributes of an object. The second level sort order gets used if the first level sort criteria is indecisive. In the list of Employees we are using as an example, there are two employees with the name Harry Major. Let us take a second sort order of age and say that in such cases we will put the employee with the younger age first.

For exactly these kinds of multiple sort ordering, Java 8 Comparator provides a default method **thenComparing()** which has the signature –

**default Comparator<T> thenComparing(Comparator<? super T> other)**

The **thenComparing()** method then does the second level sort, if the first level sort is indecisive. Let us extend the above code example to add a second-level sort criteria by age.

**Comparator's thenComparing() method for multiple sort criteria**

|  |
| --- |
| public static void main(String[] args) {    Comparator<Employee> empNameComparator = Comparator.comparing(Employee::getName).thenComparing(Employee::getAge);    Collections.sort(employeeList, empNameComparator);    employeeList.forEach(System.out::println);  } |

**OUTPUT of the above code**

Employee Name:Deborah Sprightly Age:29  
Employee Name:Ethan Hardy Age:65  
Employee Name:Harry Major Age:25  
Employee Name:Harry Major Age:35  
Employee Name:Nancy Smith Age:15  
Employee Name:Tom Jones Age:45

**Quick Explanation of the above code-**

* First **comparing()** method is invoked with method reference for **Employee**'s **getName()** method. This returns a **Comparator** instance with the first level sort based on **Employee** name as we saw in previous section.
* We append **.thenComparing(Employee::getAge)** to the **Comparator** instance returned using **comparing()** method, which adds a second level sort based on **Employee**'s **getAge()** method.
* The output is as expected with the employee named ‘Harry Major’ with the lesser age placed earlier in the sorted **employeeList** than his elder namesake employee.

**Java 8 Comparator’s natural order comparison methods**  
Java 8 Comparator supports natural order comparison of elements in a Collection. I.e. instead of defining our own comparison logic, we can instead use the inherent natural order defined for that type of element via its implementation of **Comparable** interface. Comparator provides two static methods **naturalOrder()** and **reverseOrder()** to allow natural order comparison and reverse natural order comparison respectively. These have the syntax –

**static <T extends Comparable<? super T>> Comparator<T> naturalOrder()**  
**AND**  
**static <T extends Comparable<? super T>> Comparator<T> reverseOrder()**

Let us take the case of String type which has natural comparison order defined as alphabetical. To use our existing example for sorting using Employee objects, we will extract out all the names of Employees by utilizing the **stream()** method to convert the Employee list into a Stream of Employee objects. We will then map these Employee objects using a Function into their names and collect these names into a list of Strings holding these names.  
(In case you are not aware of mapping of elements in a Stream using map() method you can read the [tutorial on Stream.map() method](https://www.javabrahman.com/java-8/java-8-mapping-with-streams-map-flatmap-methods-tutorial-with-examples/), and [tutorial on Function interface](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/).)

**Java 8 Comparator natural order sorting**

|  |
| --- |
| public static void main(String[] args) {    List<String> empNames = employeeList.stream().map(Employee::getName).collect(Collectors.toList());    empNames.sort(Comparator.naturalOrder());    empNames.forEach(System.out::println); |

**OUTPUT of the above code**

Deborah Sprightly  
Ethan Hardy  
Harry Major  
Harry Major  
Nancy Smith  
Tom Jones

**Quick Explanation of the above code-**

* Employee names are extracted into an **empNames List** as explained before the code snippet.
* empNames List is then sorted using the **Comparator.naturalOrder()** method which returns a **Comparator** instance of **String**’s natural comparison order based on the **empName**’s generic type of **String**.
* Employee names are sorted in natural comparison order, i.e. alphabetical order, and printed.
* Instead of invoking the **Comparator.naturalOrder()** method to the **empNames.sort()** method, you can invoke the **Comparator.reverseOrder()** method in order to sort the **empNames List** in reverse of natural comparison order or reverse alphabetical order.

**Java 8 Comparator’s null handling using nullsFirst()and nullsLast() methods**  
There are instances where the sort key value is nullable. We have to then decide whether we want to place objects with sort key as null earlier in the ordering than the objects with non-null sort key or at the end of the list after the objects with non-null sort key. Java 8 Comparators provide **nullsFirst()** and **nullsLast()**static methods for exactly such sort order handling of null valued sort keys. These methods have the signature –

**static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)**  
**AND**  
**static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)**

To understand the functioning of these methods, let us make the two of the employee names as null. The below two sections of code and respective output shows how the use of **nullsFirst()** and **nullsLast()** handles sorting when some objects have sort-key as null.

**Java 8 Comparator with sort key null and using nullsFirst()**

|  |
| --- |
| public class ComparatorsInJava8 {    static List<Employee> employeeList = Arrays.asList(        new Employee(null, 45),        new Employee("Harry Major", 35),        new Employee("Harry Major", 25),        new Employee(null, 65),        new Employee("Nancy Smith", 15),        new Employee("Deborah Sprightly", 29));    public static void main(String[] args) {      Comparator<Employee> empNameComparator = Comparator.comparing(Employee::getName, Comparator.nullsFirst(String::compareTo));      Collections.sort(employeeList, empNameComparator);      employeeList.forEach(System.out::println);     }  } |

**OUTPUT of the above code**

Employee Name:null Age:45  
Employee Name:null Age:65  
Employee Name:Deborah Sprightly Age:29  
Employee Name:Harry Major Age:35  
Employee Name:Harry Major Age:25  
Employee Name:Nancy Smith Age:15

Keeping the employee list same and invoking the **Comparator** using **nullLast()** –

**Java 8 Comparator with sort key null and using nullsLast()**

|  |
| --- |
| public static void main(String[] args) {      Comparator<Employee> empNameComparator = Comparator.comparing(Employee::getName, Comparator.nullsLast(String::compareTo));      Collections.sort(employeeList, empNameComparator);      employeeList.forEach(System.out::println);     }  } |

**OUTPUT of the above code**

Employee Name:Deborah Sprightly Age:29  
Employee Name:Harry Major Age:35  
Employee Name:Harry Major Age:25  
Employee Name:Nancy Smith Age:15  
Employee Name:null Age:45  
Employee Name:null Age:65

**Quick Explanation of the above code-**

* Using **nullsFirst()**, the comparator places the two **Employee** objects with the sort key null (employee name null) *before* the other Employee objects in the list.
* Likewise, **nullsLast()**, the comparator places these two **Employee** objects *after* the other Employee objects in the list.

**Java 8 Comparator’s sort order reversal method – reversed()**  
In case you simply want to sort in the order opposite to a defined comparator’s sorting order then you need not write the reverse comparison logic again. Instead simply use the **Comparator.reversed()** default method.

The **reversed()** method has the signature –

**default Comparator<T> reversed()**

**Java 8 Comparator order reversal using reversed() method**

|  |
| --- |
| public static void main(String[] args) {    Comparator<Employee> empNameComparator = Comparator.comparing(Employee::getName).reversed();    Collections.sort(employeeList, empNameComparator);    employeeList.forEach(System.out::println);  } |

**OUTPUT of the above code**

Employee Name:Tom Jones Age:45  
Employee Name:Nancy Smith Age:15  
Employee Name:Harry Major Age:35  
Employee Name:Harry Major Age:25  
Employee Name:Ethan Hardy Age:65  
Employee Name:Deborah Sprightly Age:29

**Quick Explanation of the above code-**

* Comparator’s default method **reversed()** is applied to the **empNameComparator** which has been initially defined to sort in alphabetical order of **Employee** names.
* Due to **reversed()** being applied to it, the **Employee** objects are printed in reverse alphabetical order of their names.

**Summary**– In this tutorial on Java 8 Comparators we first had a look at how Comparators were used till Java 7. Next we learnt that **Comparator** is now a functional interface and how it can be defined using a lambda expression. We then had a look at the **comparing()** method which allows us to define a Comparator using a Function instance which returns the sort key. Next we looked at **thenComparing()** method which allows us to sort using multiple sort orders. This was followed by understanding the natural order comparison related methods of Comparator – **naturalOrder()** and **reverseOrder()**. Next we looked at the null-valued sort key handling methods – **nullsFirst()** and **nullsLast()**. Lastly, we understood the **reversed()** method which allows us to sort in the reverse order of a defined **Comparator**.