Java 8

Table Of Contents

[1. Java 8 Feature 3](#_Toc512259527)

[1.1. Passing Behaviour in OOP 3](#_Toc512259528)

[1.2. Lambda’s VS Implementation 4](#_Toc512259529)

[1.3. Type inference 4](#_Toc512259530)

[1.4. Runnable using lambda 5](#_Toc512259531)

[1.5. Functional Interface 5](#_Toc512259532)

[1.6. Lambda Expression Exercise 6](#_Toc512259533)

[1.7. Using Function interfaces 10](#_Toc512259534)

[1.8. Java.util.function 12](#_Toc512259535)

[1.9. Exception Handling in Lambda’s 13](#_Toc512259536)

[1.10. Closures in lambda 13](#_Toc512259537)

[1.11. This reference in Lambda 15](#_Toc512259538)

[1.12. Method reference and Collections 16](#_Toc512259539)

[1.13. For Each Iteration 18](#_Toc512259540)

[1.14. Streams 19](#_Toc512259541)

*Java 8*

# Java 8 Feature

Here is the list of features provided by JAVA 8.

1. Enables functional programming.
2. Readable and concise code.
3. Easier to use API’s and libraries.
4. Enables support for parallel processing.

## Passing Behaviour in OOP

Prior to Java 8 in the event if the developer is to implement a function it has to be done through a class and which in turn requires instantiation the lambda expression however presents solution where in a function can be defined without encapsulating the implementation within any classes.

Example:

***Syntax 1:***

*strWelcomeMessage = () -> { System.out.println(“Hello World”);}*

In the above mentioned example the output of the computation is stored in “strWelcomeMessage” variable which can be passed as argument to another method say display. Well there is another way in which we can directly pass the lambda expression as an inline parameter.

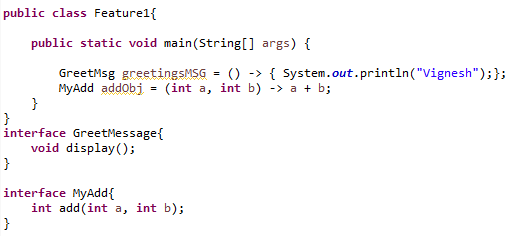
*Display(() -> {System.out.println(“Welcome”))})*

***Syntax 2:*** If the function accepts one input argument and it is also not required to write the retrun statement as well.

*Example: doubleFucntion = (int n) -> n \* 2;*

Steps to Create Lambda Expression*: The first step is to create as interface that has only one abstract method. Then the Lambda expression can be defined with signature similar to the one defined in the interface.*

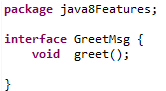
*Example :*

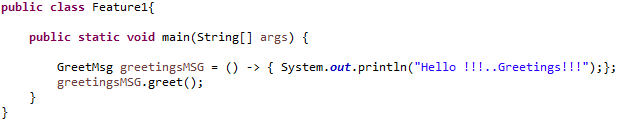


In the above mentioned example if we try to add the add method in the GreetMessage interface the corresponding lambda expression would throw an error as it is mandatory to have only one method within the interface for which the lambda is being created.

## Lambda’s VS Implementation

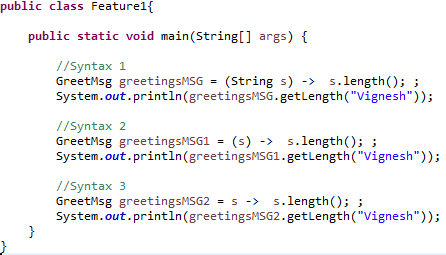
In versions prior to Java 8 for in order to execute the interface a class will be created with implementation of the interface the instance of the class will be executed. When it comes to lambda, the variable of the type interface will be created which will be initialized with a functional object and using this variable the method of the interface can be created.





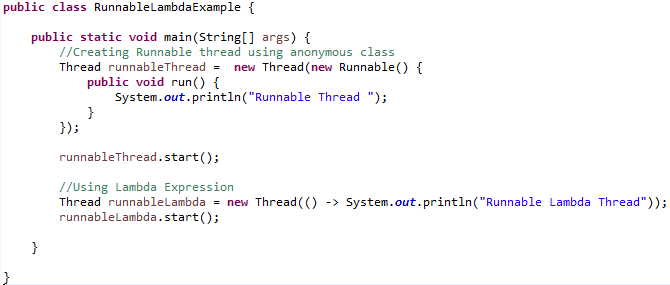
## Type inference

The java compiler performs type inference by means of which it identifies the return type and the type of the input argument sin a lambda expression is through the interface name on the left hand side.



## Runnable using lambda

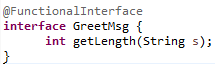
The interface such as Runnable having only one abstract method is termed as functional interface.



## Functional Interface

In order for any type to be used as validation for a lambda expression the corresponding interface should have only one abstract method and with Java 8 it is possible to have concrete method within interface then again it will not have any impact on the lambda expression as long as there is only one abstract method in the interface.

If the interface which is being used in lambda expression is being modified by adding another abstract method it would impact all the segments of the application employing lambda expression using the interface being modified. In order avoid such modification the interface can be annotated with @*FunctionalInterface* which is not mandatory but it is a good practise to let the developers know of the fact that the interface is meant for lambda expression.

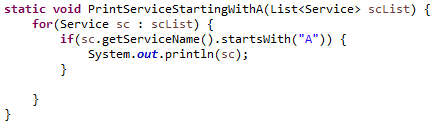


Video : 12

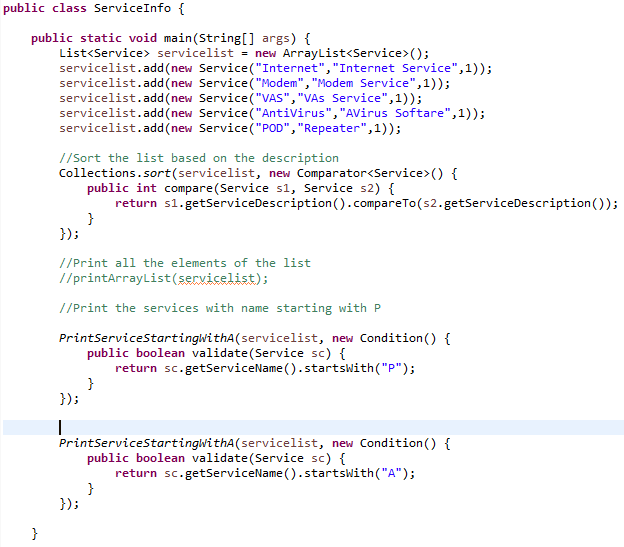
## Lambda Expression Exercise

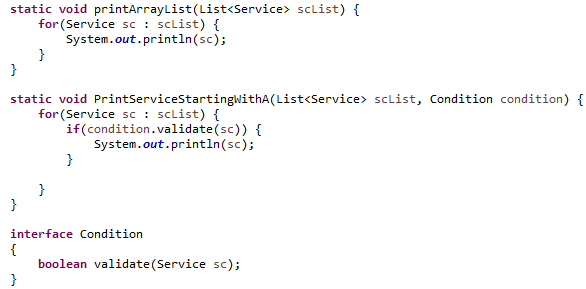




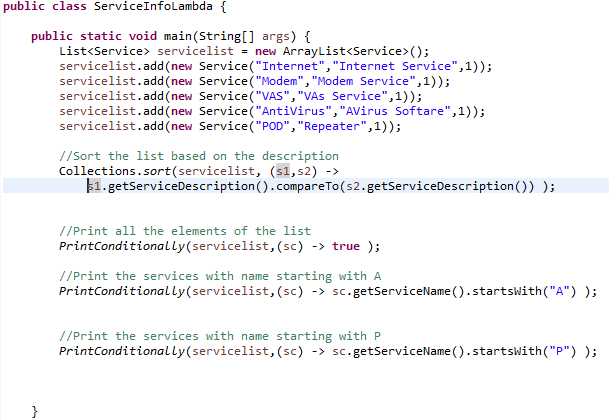


In the above mentioned class the method to print the elements with service name having the first character as “A” is too specific. In the event if the user wants to print all the service with name starting with “D”, a new method needs to be created. In order to avoid the issue lets create an interface named “Condition” which can have many implementation as per the required condition.





The above mentioned version of the code using JAVA 8.

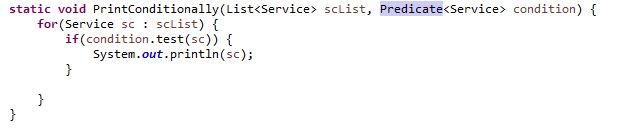


## Using Function interfaces

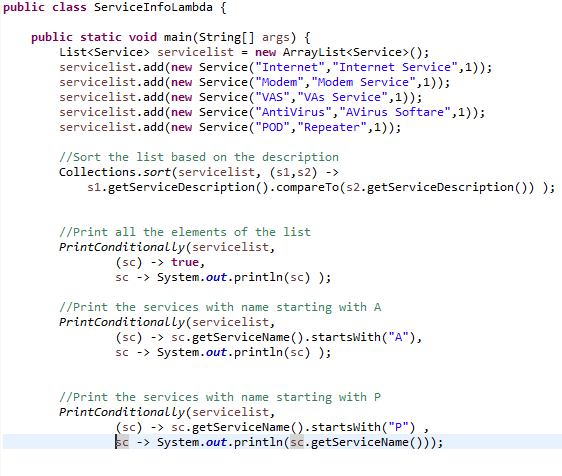
In the example described above it can be observed that the interfaces that are being used in the lambda expression contain a more generic pattern. As part of Java 8 there is a new package “java.util.function” which contain such pre-defined interface tht can be used while creating the lambda expression.

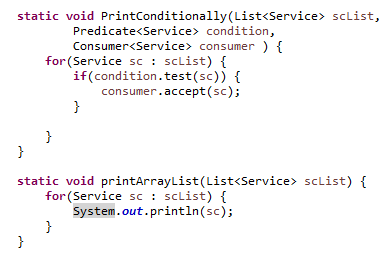
Here is the usage of one such interface named as “Predicate”. The “*PrintConditionally*” method discussed in the previous example has been modified to use the Predicate interface.

The Predicate functional interface has one method called test which accepts one argument with return type as Boolean .

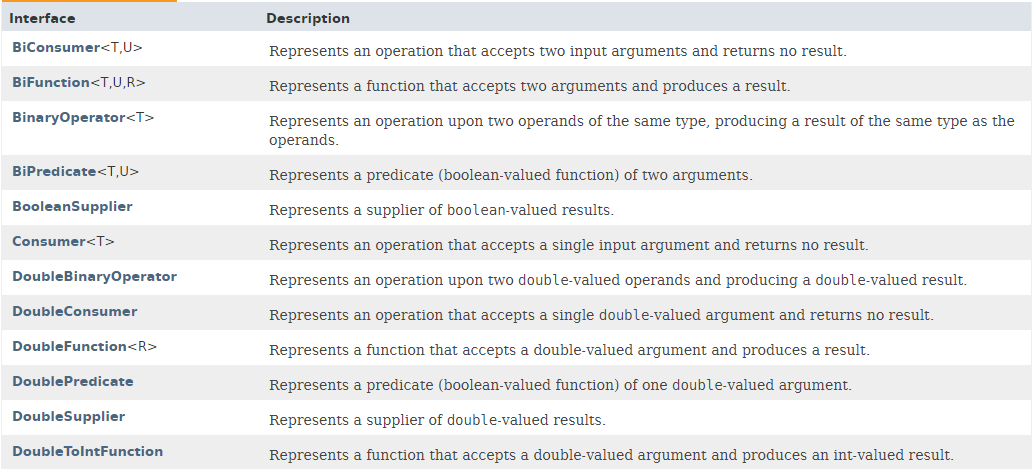


The next function interface is “Consumer” which accepts one argument with return type as void.

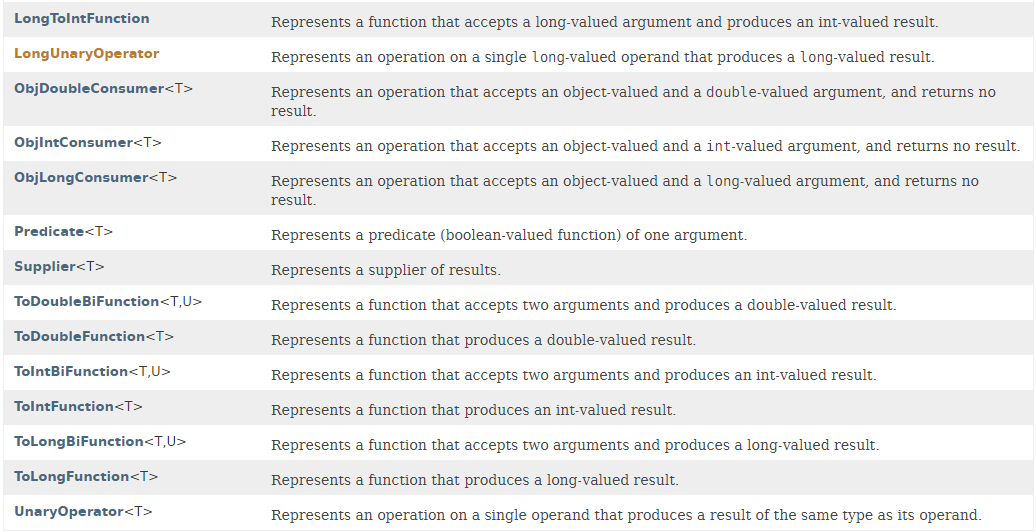




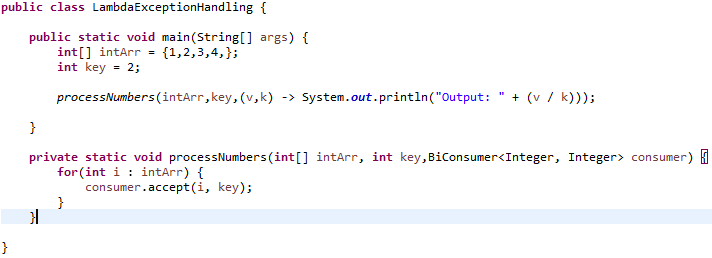
## Java.util.function





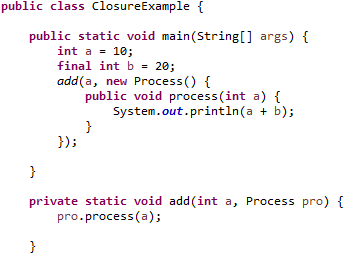


## Exception Handling in Lambda’s



## Closures in lambda

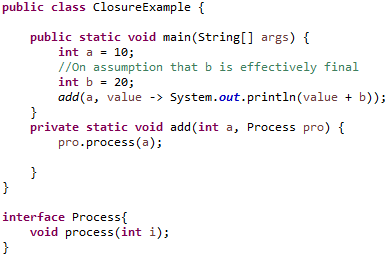
As of Java 1.7 the closures were available with a deviated syntax. The below mentioned example pictures what a closure is.



In the above mentioned example the variable “b” is defined within the scope of main method yet when add method calls upon the process method of the interface the execution is successful adding the value of a to b.

During the phase of compilation the value of the variable b is placed in the implementation of the process() method under the assumption that the value does not change during the course of execution by declaring the variable b as final.

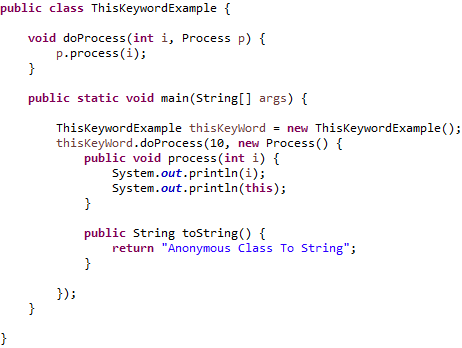
Post java 8 even if the value is not declared as final the compiler treats the variable b as effectively final and reports the modification to variable b as error throughout the application.



## This reference in Lambda

The object being referred by the “this” key word differs in value when it is used in an anonymous class versus in lambda expression.

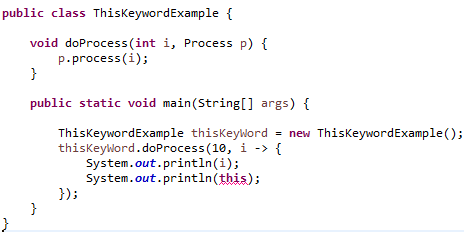
Here is an example when the “this” keyword is referred in an anonymous class.



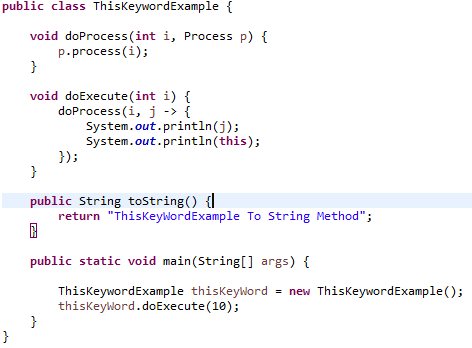
*Output:*



In the above mentioned out put the “*this*” key refers to the object of the anonymous class and not of the outer class object which is “*thisKeyWord*”. Let’s see the out value of the same in case of lambda expression.



In the above mentioned code segment the compiler reports an error stating that “*Cannot use this in static context*” reason being the reference of this key word remains unaltered within a lambda expression hence in the above mentioned segment of code the object being referred is the main class which is of type static and hence it reports an error.

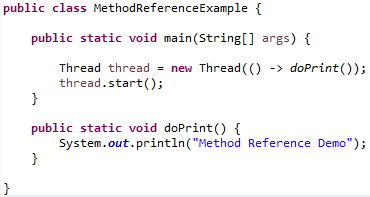


Output:

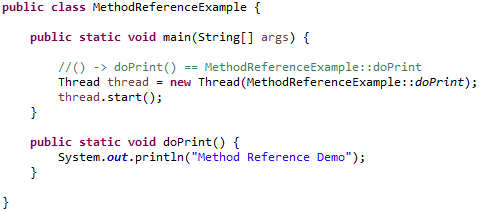


## Method reference and Collections

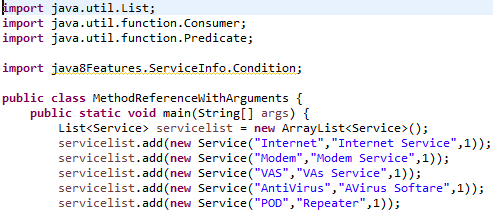
Here is the scenario where we are to execute a static method which takes in no arguments and just performs no operation with return type as void.

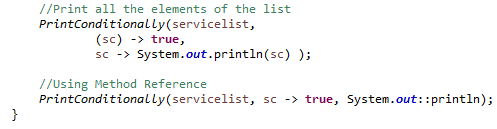


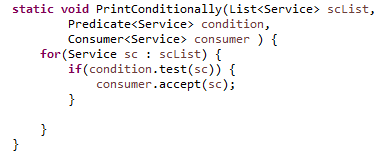
The lambda expression to call the “***doPrint***” method can be modified using method reference statement as.



The method reference can be considered as an alternative for the invocation of static method. Let’s consider another scenario where we replace a lambda expression accepting one argument with method reference.





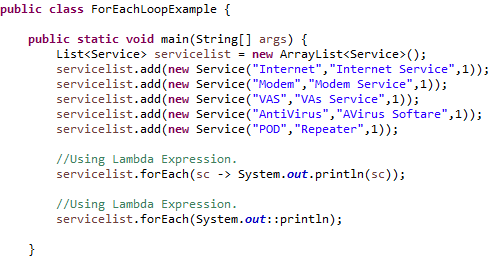


## For Each Iteration

The “FOR EACH” iteration is a new function introduced as part of Java 8 that allows us to print the contents of a list without having the conventional external iterators like “FOR LOOP” or “FOR IN LOOP”. The “FOR EACH” loop accepts one argument of type Consumer interface.

The main advantage of for each loop is that we can either pass lambda expression or Method reference as parameter.

The Consumer is an out of the box interface which accepts one argument and return void.

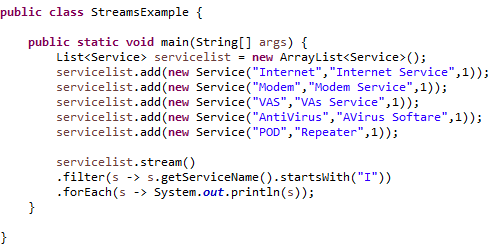


The other advantage over FOR EACH is that the order of execution is not specified it is possible for the first set of elements to be processed by Core A and the rest by another Core B which opens door for parallel processing.

## Streams

A Stream can be considered as a sequence of elements supporting parallel and sequential aggregate operations. Say we have a series of operation to be done on a list of objects in the present version of the Java we will have to start iteration and define the functionality to perform the manipulation as required. In the event if the sequence of processing is dependent on the resultant list produced by the predecessor processing, the number of iteration on the logic will also be increased.

Streams present a solution to this problem allowing the user to perform n number of operations on a list with a single traversal.



The Stream also allows the different segment of the list to be processed over distributed cores.