**JAVA 8 FEATURES**

Oracle released a new version of Java as Java 8 in March 18,2014. It was a revolutionary release of the Java for software development platform. It includes various upgrades to the Java programming, Jvm ,Tools and libraries.

**Java 8 Enhancements**

* forEach() method in Iterable Interface.
* Default and static methods in Interfaces.
* Functional Interfaces and Lambda Expressions.
* Java Stream API for bulk Data operations on collections.
* Java Time API.
* Collection API improvements.
* Concurrency API improvements.
* Java IO improvements.
* Miscellaneous core API improvements.

1. **forEach Method() in Iterable Interface**

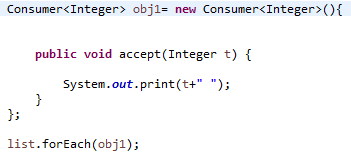
Whenever we need to traverse through a Collection, we need to create an Iterator whose whole purpose is to iterate over and then we have business logic in a loop for each of the elements in the Collection. We might get Concurrency Modification Exception if iterator is not used properly.

Java 8 has introduced *forEach* method in java.lang.Iterable interface so that while writing code we focus on business logic only. *forEach* method takes java.util.Function.Consumer object as argument, so it helps in having our business logic at a separate location that we can reuse.

The number of lines might increase but forEach method helps in having the logic for iteration and business logic at separate place resulting in higher separation of concern and cleaner code.

**External Iteration vs Internal Iteration :**

Where external iteration mixes the "what" and the "How", Internal iteration lets the client to provide only that "what" but lets the library control the "how" . This offers several potential benefits : client code can be clearer because it need only focus on stating the problem , not need only focus on stating the problem, not the details of how to go about solving it, and we can move complex optimization code into libraries where it can benefit all users.Generally, it is recommended to use internal iterator over external iterator. Internal iteration is less error prone, more readable, and requires less code.



**Java 8 Interface Changes :**

If you read forEach method details carefully, you will notice that it’s defined in Iterable interface but we know that interfaces can’t have method body. From Java 8, interfaces are enhanced to have method with implementation. We can use default and static keyword to create interfaces with method implementation.

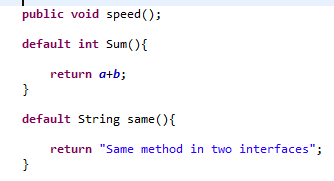
Designing interfaces have always been a tough job because if we want to add additional methods in the interfaces, it will require change in all the implementing classes. As interface grows old, the number of classes implementing it might grow to an extent that it’s not possible to extend interfaces. That’s why when designing an application, most of the frameworks provide a base implementation class and then we extend it and override methods that are applicable for our application.

**Default Method:**

For creating a default method in java interface, we need to use “**default**” keyword with the method signature.

It is not mandatory to provide implementation for default methods of interface. This feature will help us in extending interfaces with additional methods, all we need is to provide a default implementation.

A simple class that is implementing both Interface1 and Interface2 which are having default method in both of them in these case ,Its made mandatory to provide implementation for common default methods of interfaces. So if a class implementing both the above interfaces it will have to provide implementation for common method otherwise compiler will throw compile time error.+

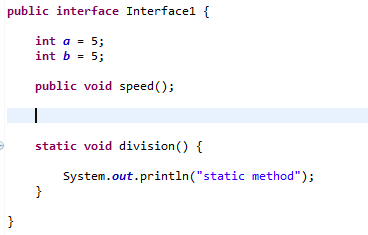


**Important points about java interface default methods:**

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
3. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it’s because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.

**Static Method :**

Java interface static method is similar to default method except that we can’t override them in the implementation classes. This feature helps us in avoiding undesired results in case of poor implementation in implementation classes.



**Important points about java interface static method:**

1. Java interface static method is part of interface, we can’t use it for implementation class objects.
2. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
3. Java interface static method helps us in providing security by not allowing implementation classes to override them.
4. We can use java interface static methods to remove utility classes such as Collections and move all of it’s static methods to the corresponding interface, that would be easy to find and use.

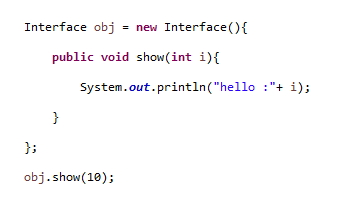
**Java 8 Functional Interface and Lambda Expression :**

Java 8, Functional Interfaces and Lambda Expressions help us in writing smaller and cleaner code by removing a lot of boiler-plate code.

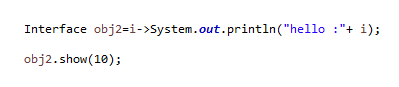
The major benefit of Java 8 functional interfaces is that we can use lambda expressions to instantiate them and avoid using bulky anonymous class implementations.

**Example :** Before Java 8 to create a instance for a Interface we can do it by using Anonymous classes but after Java 8 it is not required.

**Before Java 8:**



**After Java 8 :**

****

So all the bulky boiler plate code is reduced after Java 8 .

**Functional Interface :**

An Interface with exactly one abstract method is called Functional Interface.

@Functional Interface annotation is added so that we can mark an interface as functional interface.

Java 8 has defined a lot of functional interfaces in java.util.function package.

Some of the useful java 8 functional interfaces are:

* Consumer
* Function
* Predicate
* Binary Operator
* Supplier

**Supplier interface** represents a function which does not take in any argument but produces a value of type T.

**Consumer interface** is a functional interface that represents an function that consumes a value without returning any value. A Java Consumer implementation could be printing out a value, or writing it to a file, or over the network etc.

**Predicate interface**, java.util.function.Predicate, represents a simple function that takes a single value as parameter, and returns true or false.

**Lambda Expression:**

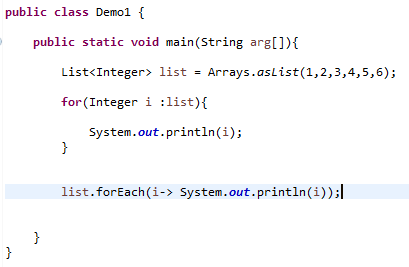
In very simple terms, a lambda expression is a function that can be refer and passed around a object.

The Lambda Expression is used to provide the implementation of Functional interface . It saves a lot of code. In case of lambda expression we don't need to define the method again for providing the implementation.

Using Lambda Expression we can create instance of a functional interface using lambda expression rather than using anonymous class.

**Why do we need Lambda Expression?**

* Reduced lines of code.
* Sequential and Parallel Execution Support.



**Java 8 Stream**

Java provides a new additional package in Java 8 called java.util.stream. This package consists of classes, interfaces and enum to allows functional - style operations on the elements you can use stream by importing java.util.stream package.

Using stream API to implement internal iteration , that is better because java framework is in control of the iteration.

Internal iteration provides several features such as sequential and parallel execution , filtering based on the given criteria , mapping.

**Stream Features :**

* Streams does not store elements . It simply conveys elements from a source such as a data structure.
* Stream is functional in nature. Operations performed on stream does not modify its source.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream.

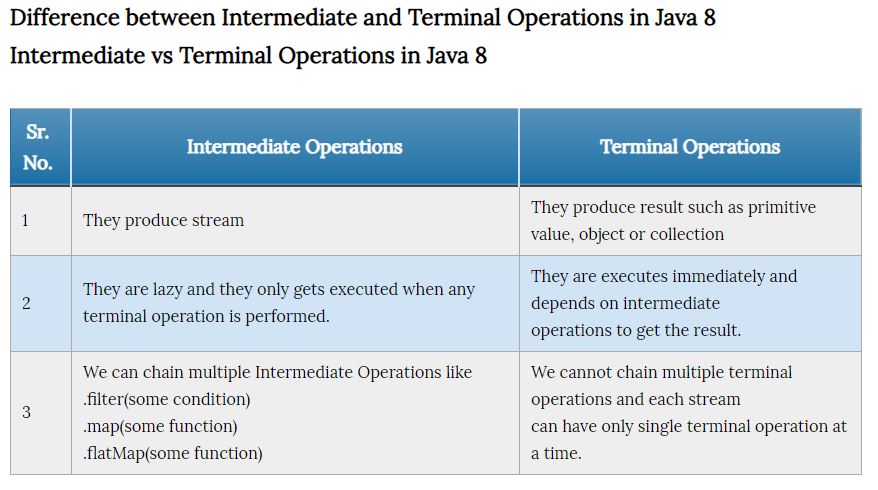
**Intermediate and Terminal Operations :**

Intermediate operation is a operation that adds a listener to the stream without doing anything else.

Terminal operation is an operation that starts the internal iteration of the elements, call all the listeners, and returns result.

**Stream Lazy Evaluation :**

Each intermediate operation creates a new stream but does not iterate elements and performs operations until the terminal operation is invoked.

****

**Stream Short Circuiting Operations :**

* An Intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream. For example limit and skip are two short circuiting intermediate operations.
* A Terminal operation is called short circuiting if it may terminate in finite time for infinite stream. For example anyMatch, allMatch, noneMatch, findFirst and find any are short circuiting terminal operations.

**Intermediate Operations :**

* Filter()
* Map()
* flatMap()
* Distinct()
* Limit()
* Peek()

**Terminal Operations :**

* anyMatch()
* allMatch()
* noneMatch()
* Collect()
* Count()
* findAny()
* findFirst()
* forEach()
* Min()
* Max()
* Reduce()
* toArray()

**Java Time API:**

It has always been hard to work with Date, Time and Time Zones in java. There was no standard approach or API in java for date and time in Java. One of the nice addition in Java 8 is the java.time package that will streamline the process of working with time in java.

Just by looking at Java Time API packages, I can sense that it will be very easy to use. It has some sub-packages java.time.format that provides classes to print and parse dates and times and java.time.zone provides support for time-zones and their rules.

The new Time API prefers enums over integer constants for months and days of the week. One of the useful class is DateTimeFormatter for converting datetime objects to strings.

**Method Reference :**

A method reference is a Java 8 construct that can be used for referencing a method without invoking it. It is used for treating methods as Lambda Expressions . They only work as syntactic sugar to reduce verbosity of same lambdas.