**Java 1.8**

**Java 8 Programming Language Enhancements**

Java 8 provides following features for Java Programming:

* Lambda expressions,
* Method references,
* Functional interfaces,
* Stream API,
* Default methods,
* Base64 Encode Decode,
* Static methods in interface,
* Optional class,
* Collectors class,
* ForEach() method,
* Parallel array sorting,
* Nashorn JavaScript Engine,
* Parallel Array Sorting,
* Type and Repating Annotations,
* IO Enhancements,
* Concurrency Enhancements,
* JDBC Enhancements etc.

**default methods:** Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

**=================================================**

**Java forEach loop**

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

**forEach() Signature in Iterable Interface**

**default** **void** forEach(Consumer<**super** T>action)

## Java Stream forEachOrdered() Method

Along with forEach() method, Java provides one more method forEachOrdered(). It is used to iterate elements in the order specified by the stream.

### Singnature: void forEachOrdered(Consumer<? super T> action)

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ForEachLoop {

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

List<String> gamesList = **new** ArrayList<String>();

gamesList.add("Football");

gamesList.add("Cricket");

gamesList.add("Chess");

gamesList.add("Hocky");

System.***out***.println("------------Iterating by Lamda Expressin---------------");

gamesList.forEach(games -> System.***out***.println(games));

System.***out***.println("------------Iterating by passing method reference---------------");

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

System.***out***.println("------------Iterating by passing method reference---------------");

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

System.***out***.println("------------Iterating by Lamda Expressin---------------");

gamesList.stream().forEachOrdered(game->System.***out***.println(game));

}

}

flatMap() V/s map() :

1) map() takes a Stream and transform it to another Stream. It applies a function on each element of Stream and store return value into new Stream. It does not flatten the stream. But flatMap() is the combination of a map and a flat operation i.e, it applies a function to elements as well as flatten them.

2) map() is used for transformation only, but flatMap() is used for both transformation and flattening.

class GFG

{

// Driver code

public static void main(String[] args)

{

// Creating a list of Prime Numbers

List<Integer> PrimeNumbers = Arrays.asList(5, 7, 11,13);

// Creating a list of Odd Numbers

List<Integer> OddNumbers = Arrays.asList(1, 3, 5);

// Creating a list of Even Numbers

List<Integer> EvenNumbers = Arrays.asList(2, 4, 6, 8);

List<List<Integer>> listOfListofInts =

Arrays.asList(PrimeNumbers, OddNumbers, EvenNumbers);

System.out.println("The Structure before flattening is : " +

listOfListofInts);

// Using flatMap for transformating and flattening.

List<Integer> listofInts = listOfListofInts.stream()

.flatMap(list -> list.stream())

.collect(Collectors.toList());

System.out.println("The Structure after flattening is : " +

listofInts);

}

}