Java 1.8

java 10 is going to release on september 2018 with new features of

JSR 354: Money and Currency API

|  |  |  |
| --- | --- | --- |
| **Version** | **Release date** | **End of Public Updates** |
| Java SE 8  Java SE 9 | 2014  2017 | March  March 2018 |
| Java SE 10 (18.3) | 2018 | September 2018 |
| Java SE 11 (18.9 LTS) | 2018 | N/A |

* [default and static methods in Interfaces](https://www.journaldev.com/2389/java-8-features-with-examples#interface-default-static-method)
* [Functional Interfaces and Lambda Expressions](https://www.journaldev.com/2389/java-8-features-with-examples#functional-interface-lambdas)
* [Java Stream API for Bulk Data Operations on Collections](https://www.journaldev.com/2389/java-8-features-with-examples#java-stream-api)
* [Java Time API](https://www.journaldev.com/2389/java-8-features-with-examples#java8-time)
* [Collection API improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-collection)
* [Concurrency API improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-concurrency)
* [Java IO improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-io)
* [Miscellaneous Core API improvement](https://www.journaldev.com/2389/java-8-features-with-examples#java8-core)s
* [forEach() method in Iterable interface](https://www.journaldev.com/2389/java-8-features-with-examples#iterable-forEach)

1.[default and static methods in Interfaces](https://www.journaldev.com/2389/java-8-features-with-examples" \l "interface-default-static-method):-

In java 1.8 we have facilities of implemented method of type static and default inside interface.

ForEach() is an example of default method which created inside of Iterable interface.

default void forEach(Consumer<? super T> action) {

Objects.requireNonNull(action);

for (T t : this) {

action.accept(t);

}

}

**Differences between static and default methods in Java 8:**

1) Default methods **can be** overriden in implementing class, while static **cannot**.

2) Static method belongs **only** to Interface class, so you can only invoke static method on Interface class, not on class implementing this Interface, see:

3) Both class and interface **can have** static methods with same names, and neither overrides other!.

4.in case of multiple interface ,two interface can have same method in this case class have to override that method but in this case we can lose code given by the interface if we don't want to lose the existing code given by the interface then in this case we use the code like this

Interface1.super.log("vicky");

here method will be called from Interface1  
e.g.

package com.hcl.clazz;

import com.hcl.in.Interface1;

import com.hcl.in.Interface2;

public class MyClaas implements Interface1, Interface2 {

@Override

public void method2() {

}

@Override

public void method1(String str) {

}

//MyClass won't compile without having it's own log() implementation

@Override

public void log(String str){

System.out.println("MyClass logging::"+str);

Interface1.super.log(str)//it will call the log method of interface1 so existing code will be used with implementing or redefine the code.

Interface1.print("abc");

}

**public** **static** **void** print(String str){

System.***out***.println("in MyClass "+str);

}

}

**package** com.hcl.in;

@FunctionalInterface

**public** **interface** Interface2 {

**void** method2();

**default** **void** log(String str){

System.***out***.println("I2 logging::"+str);

}

}

**package** com.hcl.in;

**public** **interface** Interface1 {

**void** method1(String str);

**default** **void** log(String str){

System.***out***.println("I1 logging::"+str);

}

**static** **void** print(String str){

System.***out***.println("Printing "+str);

}

//trying to override Object method gives compile time error as

//"A default method cannot override a method from java.lang.Object"

// default String toString(){

// return "i1";

// }

}

**Advantages of default and static method**

java 8 uses **default** and **static** methods heavily in [**Collection API**](https://www.journaldev.com/1260/collections-in-java-tutorial) and default methods are added so that our code remains backward compatible.

**Why we need of default method in java 1.8**

In java 1.8 ,help us to add new functionality without breaking contract between classes and interface.

For example

Suppose we have an Expression interface that has ConstantExpression, BinaryExpression, DivisionExpression etc. as existing implementations. Now we get a requirement to add new functionality.

Return the signum of the evaluated result.

Return signum after evaluating the expression.

This can be done with default and static methods without breaking any functionality as follows.

public interface Expression {

double evaluate();

default double signum() {

return signum(evaluate());

}

static double signum(double value) {

return Math.signum(value);

}

}

**Default Methods and Multiple Inheritance Ambiguity Problems:-**

Java supports multiple inheritance of interfaces. Consider having two interfaces, InterfaceA and InterfaceB , having default methods with the same signature. Your class ConcreteC is implementing both the interfaces.

interface InterfaceA {

void performA();

default boolean canPerform() {

// return true if I can perform the action

}

}

interface InterfaceB {

void performB();

default boolean canPerform() {

//return true if I can perform the action

}

class ConcreteC implements InterfaceA, InterfaceB {

}

The above code will fail to compile with "error: unrelated defaults for canPerform()from InterfaceA and InterfaceB."

To overcome this problem, you need to override the default method.

class ConcreteC implements InterfaceA, InterfaceB {

override

public boolean canPerform() {

}

}

But say you don't want to provide the implementation of the overridden default method but instead want to reuse the existing one. That is also possible with the following syntax.

class ConcreteC implements InterfaceA, InterfaceB {

override

public boolean canPerform() {

return InterfaceA.super.canPerform();

}

}

2.**[forEach() method in Iterable interface](https://www.journaldev.com/2389/java-8-features-with-examples" \l "iterable-forEach):-**

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 ConcurrentModificationException if iterator is not used properly.

This problem can be override in java 1.8 because ,forEach() has introduced in java.lang.Iterable interface which takes care about ConcurrentModificationException.Now we have to concentrate on only business logic while writing a code because forEach() takes a parameter as java.util.function.consumer object which helps in having our business logic at seprate location.

How forEach() handle the ConcurrentModificationException?

To know this we need to check internal implementationof forEach() method

If we wandering , how iterator checks for modification, we can check its implementation in AbstractList class.

In AbstractList class,there is one variable modCount which provides how many times a List has modified in each call of next() by calling checkForCoModification() method.

For e.g.

**package** java8foreachexample;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

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

**public** **class** Java8ForEachexample {

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

//creating sample Collection

List<Integer> myList = **new** ArrayList<Integer>();

**for**(**int** i=0; i<10; i++) myList.add(i);

//traversing using Iterator

Iterator<Integer> it = myList.iterator();

**while**(it.hasNext()){

Integer i = it.next();

System.***out***.println("Iterator Value::"+i);

}

//traversing through forEach method of Iterable with anonymous class

myList.forEach(**new** Consumer<Integer>() {

**public** **void** accept(Integer t) {

System.***out***.println("forEach anonymous class Value::"+t);

}

});

}

}

output:-

Iterator Value::0

Iterator Value::1

Iterator Value::2

Iterator Value::3

Iterator Value::4

Iterator Value::5

Iterator Value::6

Iterator Value::7

Iterator Value::8

Iterator Value::9

forEach anonymous class Value::0

forEach anonymous class Value::1

forEach anonymous class Value::2

forEach anonymous class Value::3

forEach anonymous class Value::4

forEach anonymous class Value::5

forEach anonymous class Value::6

forEach anonymous class Value::7

forEach anonymous class Value::8

forEach anonymous class Value::9

**Q.)what is ConcurrentModificationException?**

Java.util.ConcurrentModificationException can occure in multithreaded and single threaded application.

ConcurrentModificationException can occure while working with Collection classes. Java collection classes are fail-fast which means while traversing of element from collection uisng iterator , Iterator.hasNext() will through ConcurrentModificationException if and only if ,element of collection get changed by thread during travering process.

Concurrent Modification exception scenario

**package** concurrentmodificationexample;

**import** java.util.\*;

**public** **class** IteratorExample {

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

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

myList.add("1");

myList.add("2");

myList.add("3");

myList.add("4");

myList.add("5");

Iterator<String> it = myList.iterator();

**while**(it.hasNext()){

String value = it.next();

System.***out***.println("List Value:"+value);

**if**(value.equals("3")) myList.remove(value);

}

Map<String,String> myMap = **new** HashMap<String,String>();

myMap.put("1", "1");

myMap.put("2", "2");

myMap.put("3", "3");

Iterator<String> it1 = myMap.keySet().iterator();

**while**(it1.hasNext()){

String key = it1.next();

System.***out***.println("Map Value:"+myMap.get(key));

**if**(key.equals("2")){

myMap.put("1","4");

//myMap.put("4", "4");

}

}

}

}

Above program will throw java.util.ConcurrentModificationException when executed, as shown in below console logs.

List Value:1

List Value:2

List Value:3

Exception in thread "main" java.util.ConcurrentModificationException

at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:907)

at java.util.ArrayList$Itr.next(ArrayList.java:857)

at concurrentmodificationexample.IteratorExample.main(IteratorExample.java:18)

If we wandering , how iterator checks for modification, we can check its implementation in AbstractList class.

In AbstractList class,there is one variable modCount which provides how many times a List has modified in each call of next() by calling checkForCoModification() method.

**Avoid ConcurrentModificationException in Single Thread and multi-thread.**

**Multithread**

1.convert the list into array and then iterate the array .This approach is best suitable for the smaller size of list if List size is more then it reduces the performance of application.

2.We can use synchronize block while iterating the list but its not recommended beause it cease the benefits of multithreading concept.

3.if we are using java 1.5 or higer then we can use ConcurrentHashMap in case and CopyOnWriteArrayList classes .This approach is recommended.

**Single Thread.**

If you are working on single-threaded environment and want your code to take care of the extra added objects in the list then you can do so using for loop rather than iterator.

for(int i = 0; i<myList.size(); i++){

System.out.println(myList.get(i));

if(myList.get(i).equals("3")){

myList.remove(i);

i--;

myList.add("6");

}

}

**package** concurrentmodificationexample;

**import** java.util.\*;

**import** java.util.concurrent.ConcurrentHashMap;

**import** java.util.concurrent.CopyOnWriteArrayList;

**public** **class** IteratorExample {

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

List<String> myList = **new** CopyOnWriteArrayList<String>();

myList.add("1");

myList.add("2");

myList.add("3");

myList.add("4");

myList.add("5");

Iterator<String> it = myList.iterator();

**while**(it.hasNext()){

String value = it.next();

System.***out***.println("List Value:"+value);

//if(value.equals("3")) myList.remove(value);

}

Map<String,String> myMap = **new** ConcurrentHashMap<String,String>();

myMap.put("1", "1");

myMap.put("2", "2");

myMap.put("3", "3");

Iterator<String> it1 = myMap.keySet().iterator();

**while**(it1.hasNext()){

String key = it1.next();

System.***out***.println("Map Value:"+myMap.get(key));

**if**(key.equals("2")){

myMap.put("1","4");

myMap.put("4", "4");

}

}

}

}

output

List Value:1

List Value:2

List Value:3

List Value:4

List Value:5

Map Value:1

Map Value:2

Map Value:3

Map Value:4

From the above example its clear that:

1.Concurrent Collection classes can be modified safely, they will not throw ConcurrentModificationException.

2.In case of CopyOnWriteArrayList, iterator doesn’t accommodate the changes in the list and works on the original list.

**3.**[Functional Interfaces and Lambda Expressions](https://www.journaldev.com/2389/java-8-features-with-examples" \l "functional-interface-lambdas)**:-**

Functional interface has introduces in java 1.8 .An interface has exactly one abstract method become as Functional interface.java.lang.Runnable interface with single abstract method is an example of functional interface.

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression.

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

One of the benefit of Functional interface to use Lambada expression to initiate them.

Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

Java Lambda Expression Syntax

1. (argument-list) -> {body}
2. ava lambda expression is consisted of three components.
3. **1) Argument-list:** It can be empty or non-empty as well.
4. **2) Arrow-token:** It is used to link arguments-list and body of expression.
5. **3) Body:** It contains expressions and statements for lambda expression.
6. Let's see a scenario. If we don't implement Java lambda expression. Here, we are implementing an interface method without using lambda expression.

We can give the implementation for the single method interface using anonymous class but code looks like bulky. Let c the example given below.

Interface Drawable{

Public void draw();

}

Class MyClass implement Drawable{

Public static void main()String args[]){

Int width =10;

Drawable d = new Drawable(){

@override

Public void draw(){

System.out.println(width);

}};

d.draw();

}

}

Now see the same program using Lambda expression

Interface Drawable{

Public void draw();

}

Class MyClass implement Drawable{

Public static void main(Str)ing args[]){

Int width 10;

Drawable d=()🡪{

System.out.println(width);

}

d.draw();

}

}

We can initiate interface with anonymous class also but its look like bulky.

Runnable r = new Runnable(){

@Override

public void run() {

System.out.println("My Runnable");

}};

To initiate the Interface using Lambda expression we use only method argument and business logic. We can write above code using Lambda expression given below.

Runnable r1 =()→{

sysout(“My Runnable”);

{

If you have single statement in method implementation, we don’t need curly braces also. For example above Interface1 anonymous class can be instantiated using lambda as follows.

Interface1 i1 = (s) -> System.out.println(s);

i1.method1("abc");

So lambda expressions are means to create anonymous classes of functional interfaces easily. There are no runtime benefits of using lambda expressions, so I will use it cautiously because I don’t mind writing few extra lines of code.

A new package java.util.function has been added with bunch of functional interfaces to provide target types for lambda expressions and method references.

**4.[Java Stream API for Bulk Data Operations on Collections](https://www.journaldev.com/2389/java-8-features-with-examples" \l "java-stream-api)**

In java 8 java.util.stream has been added to work on collection for filter/reduce/map kind of operation based on some condition.

Stream Api allow us to perform sequential and parallel operation. [Install](http://marketplace.eclipse.org/marketplace-client-intro?mpc_install=84997)

Collection interface has extended with stream() and parallelStream() default methods to get stream on sequential and parallel execution.

Example

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

public class StreamExample {

public static void main(String[] args) {

List<Integer> myList = new ArrayList<>();

for(int i=0; i<100; i++) myList.add(i);

//sequential stream

Stream<Integer> sequentialStream = myList.stream();

//parallel stream

Stream<Integer> parallelStream = myList.parallelStream();

//using lambda with Stream API, filter example

Stream<Integer> highNums = parallelStream.filter(p -> p > 90);

//using lambda in forEach

highNums.forEach(p -> System.out.println("High Nums parallel="+p));

Stream<Integer> highNumsSeq = sequentialStream.filter(p -> p > 90);

highNumsSeq.forEach(p -> System.out.println("High Nums sequential="+p));

}

}

Here parallel input is not in order manner but in sequential output is in order.

5.[Java Time API](https://www.journaldev.com/2389/java-8-features-with-examples" \l "java8-time):-

in java 8,java.time packages has added which streamline the process of working with time in java.

In time packages if we see the sub-package java.time.zone which provides the time zone and their rules.

Java.time.format ,which provides facilities to parse date and time.

6.Concurrency API improvements

Some important concurrent API enhancements are:

* ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
* CompletableFuture that may be explicitly completed (setting its value and status).
* Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

**7.Java IO improvements**

Some IO improvements known to me are:

* Files.list(Path dir) that returns a lazily populated Stream, the elements of which are the entries in the directory.
* Files.lines(Path path) that reads all lines from a file as a Stream.
* Files.find() that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
* BufferedReader.lines() that return a Stream, the elements of which are lines read from this BufferedReader.