**Lambda Stream API**

**Lambda Expressions:** Lambda expressions can be used to facilitate functional programming and simplifies development a lot.

* If you have an interface with one method, instead of creating object we can use lambda expression and eliminate boiler plate code.
* We use lambda expressions only instead of anonymous inner class with one method.

A lambda expression is characterized by the following syntax.

parameter -> expression body

Following are the important characteristics of a lambda expression.

**Optional type declaration** − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.

**Optional parenthesis around parameter** − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.

**Optional curly braces** − No need to use curly braces in expression body if the body contains a single statement.

**Optional return keyword** − The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

**Functional Programming:** Defining the body of the method at the time of developing the application. Lambda expressions provide functional programming. We can use lambda instead of Anonymous inner class.

**Stream:** It is the new abstract layer introduced in Java 8. We can use Stream API to process data in declarative way, like SQL statements.

SQL statements directly return the result without doing any computation at the developer end.

Using collections framework in java, developer has to **Use Loops(External Iterations).** Another concern is efficiency, developer has to write parallel code processing using multi-core processors which can be pretty error prone. To resolve such issues, Java 8 introduced the concept of Stream that lets the developer process data declaratively and leverage multicore architecture without need to write any specific code for it.

**Stream represents a sequence of objects, which supports aggregate operations like filter, map, limit, reduce, find, match and so on.**

Stream provides a set of elements of specific type in a sequential manner, it computes the elements if specified

Stream takes collections, Arrays, I/O resources as input source.

Stream operations can **do the iterations internally** on the source of elements.

**Pipelining**: we can do pipelining in stream operations using collect() method.

**Generating Streams:**

* **stream()** − Returns a sequential stream considering collection as its source. It returns object of stream interface.
* **parallelStream()** − Returns a parallel Stream considering collection as its source. It uses systems cores and perform parallel processing using multi-threading.

**forEach:** Is a new method introduced in Java 8 which takes the object of consumer Interface(FI) Single Abstract method Interface(SAM).

This concept is called Internal Iterations.

A screenshot of a social media post

Description automatically generated

**Method References:**

Method references help to point to methods by their names. A method reference is described using **"::"** symbol. A method reference can be used to point the following types of methods − Static methods, Instance methods and Constructors using new operator (TreeSet::new)

**A screenshot of a social media post

Description automatically generated**

**Functional Interfaces:**

All methods here takes generic input parameter.

1. Consumer Interface: accept(Integer i) 🡺 forEach takes consumer object.
2. Function Interface: apply(Integer i) 🡺 map method takes function object.
3. BinaryOperator: apply(Integer i) 🡺 reduce method takes binaryOperator object.
4. Predicate: test(Integer i) 🡺 filter method takes object of predicate Interface

**Methods of Stream API:**

Stream cannot be reused, just can be used to process large amounts of data.

1. **Map():** needs an object of Function.

**A screenshot of a social media post

Description automatically generated**

1. **reduce():** needs an object of Binary Operator.

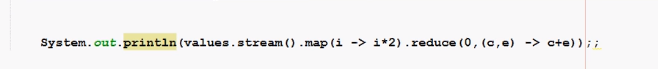
**A screenshot of a social media post

Description automatically generated**

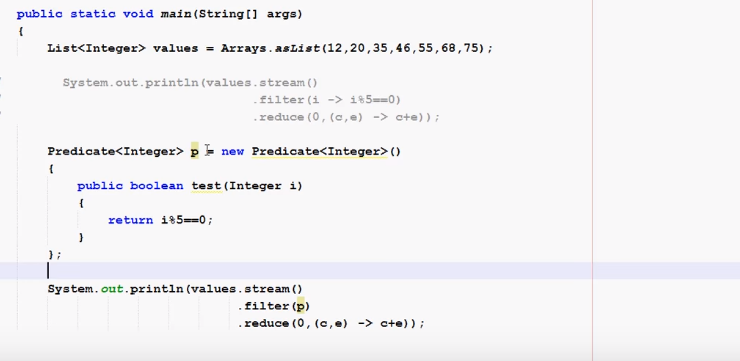
**Using lambda for the above:**

**A screenshot of a cell phone

Description automatically generated**

****

1. **filter():** needs an object of predicate.

****

**A screenshot of a cell phone

Description automatically generated**

* **filter, map are lazy methods,** they directly don’t perform the operation. They will talk to consecutive methods and upon the requirement they will execute.
* In the below example filter and map will not perform operation for all the values of stream. They just perform operation for the first value of stream.

1. **findFirst():** will return the first element of the stream, which is **optional** value.
2. **orElse():** used to get output specified if no result returned.

**A screenshot of a cell phone

Description automatically generated**

**Lazy Execution of filter and map methods:**

**A screenshot of a cell phone

Description automatically generated**

Inside methods the output here is not all stream of values. Its just the first element. It is more efficient than normal for loop.

A screenshot of a cell phone

Description automatically generated

1. **limit:** The ‘limit’ method is used to reduce the size of the stream.

Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

1. **sorted:** The ‘sorted’ method is used to sort the stream.

Random random = new Random();

random.ints().limit(10).sorted().forEach(System.out::println);

1. **Collectors**: collectors are used to combine the result of processing on the elements of stream. Collectors can be used to return a list or a string.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.**toList**());

System.out.println("Filtered List: " + filtered);

String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.**joining**(", "));

System.out.println("Merged String: " + mergedString);

1. **Statistics**: statistics collectors are also introduced to calculate all statistics when stream processing is being done.

List numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

**IntSummaryStatistics** stats = numbers.stream().mapToInt((x) -> x).**summaryStatistics**();

System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());

**Optional:** will reduce the bugs in program by giving **optional.empty** when there is no value.

* Helps the developers to handle NPE and can perform some predefined actions instead of throwing NPE.

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

**Date and Time API:**

The old date API of java.util.Date has drawbacks such as

1. **Not thread safe**: developer has to deal with the concurrency issues. The new date-time API is **thread safe**, immutable and doesn’t have setter methods.
2. **Poor design**: no uniformity between months zero based and days 1 based and less direct methods. New API has numerous utility methods for such operations.
3. **Difficult time Zone handling**: developers had to write a lot of code for time Zone issues. New API has been developed resolving all these **by developing API as ISO- centric with many domains.**

Java 8 introduces new date time API under **java.time** package. Some of important classes introduced are

1. **Local**: Simplified date-time API with no complexity of timezone handling.
   1. **LocalDate, LocalTime, LocalDateTime**
2. **Zoned**: Specialized date-time API to deal with various timezones.

**LocalDate:** used to represents date in ISO format(yyyy-MM-dd) without time.

* **Using now() method:**

A screenshot of a cell phone

Description automatically generated

* **Using of() and parse() methods:**

**A screenshot of a social media post

Description automatically generated**

* we can plus/minus days, months or years to given date.
* minus takes unit input from **ChronoUnit** Enumeration.

**A screenshot of a social media post

Description automatically generated**

* can get DayOfWeek and month from given date.
* can check a year is leap or not.

**A screenshot of a cell phone

Description automatically generated**

* can compare two dates.

**A screenshot of a cell phone

Description automatically generated**

**LocalTime:** represents only time without date. Can get local time object using now(), parse() and of() methodA screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**LocalDateTime:** used to represent combination of date and time.

A screenshot of a cell phone

Description automatically generated

* there are many utility methods to add and subtract hours and months etc..

A screenshot of a social media post

Description automatically generated

**ZonedDateTime:** we use this when we need to deal with specific time zoned date and time. ZoneId is a identifier used to represent different zones.

* We can convert time from specific Zone to other using ZonedDateTime.
* We can also convert datetime using offset values manually using OffsetDateTime.

A screenshot of a social media post

Description automatically generated

**Period and Duration:** period class represents the quantity of time in years, months and days. Duration class represents a quantity of time in terms of seconds and nano seconds.

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Date and Time Formatting:**

**A screenshot of a cell phone

Description automatically generated**

**Nashorn JavaScript:** Java 8 introduced a much-improved JavaScript engine called Nashorn to provide better performance. It directly compiles the code in memory and passes the byte code to JVM.

JJS: Command line tool to execute JS

**Interpreting js File**

Create and save the file **sample.js** in c:\> JAVA folder.

### **sample.js**

print('Hello World!');

Open console and use the following command.

C:\JAVA>jjs sample.js

It will produce the following output:

Hello World!

### **jjs in Interactive Mode**

Open the console and use the following command.

C:\JAVA>jjs

jjs> print("Hello, World!")

Hello, World!

jjs> quit()

>>

## Calling JavaScript from Java

Using ScriptEngineManager, JavaScript code can be called and interpreted in Java.

### **Example**

Create the following Java program using any editor of your choice in, say, C:\> JAVA.

**Java8Tester.java**

[Live Demo](http://tpcg.io/9wGcC5)

import javax.script.ScriptEngineManager;

import javax.script.ScriptEngine;

import javax.script.ScriptException;

public class Java8Tester {

public static void main(String args[]) {

ScriptEngineManager scriptEngineManager = new ScriptEngineManager();

ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");

String name = "Mahesh";

Integer result = null;

try {

nashorn.eval("print('" + name + "')");

result = (Integer) nashorn.eval("10 + 2");

} catch(ScriptException e) {

System.out.println("Error executing script: "+ e.getMessage());

}

System.out.println(result.toString());

}

}

### **Verify the Result**

Compile the class using **javac** compiler as follows −

C:\JAVA>javac Java8Tester.java

Now run the Java8Tester as follows −

C:\JAVA>java Java8Tester

It should produce the following result −

Mahesh

12

## Calling Java from JavaScript

The following example explains how to import and use Java classes in java script.

Create and save **sample.js** in c:\> JAVA folder.

### **sample.js**

var BigDecimal = Java.type('java.math.BigDecimal');

function calculate(amount, percentage) {

var result = new BigDecimal(amount).multiply(new BigDecimal(percentage)).divide(

new BigDecimal("100"), 2, BigDecimal.ROUND\_HALF\_EVEN);

return result.toPlainString();

}

var result = calculate(568000000000000000023,13.9);

print(result);

Open the console and use the following command.

C:\JAVA>jjs sample.js

It should produce the following output −

78952000000000000003.20

Old Notes:

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* Stream API helps in achieving functional programming and concurrency. By this we can focus on “what to do” instead of “how to do it”.
* Instead of using external iterations, we can use internal iterations with stream API(forEach).
* Values.forEach(i -> sout(i));
* We need to give consumer inerface to forEach method.
* forEach method is a default method in the interface came in java 1.8.
* we can pass method reference as object to forEach method.
* Values.forEach(System.out::println);

f

**public** **class** foreachimple {

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

List<Integer> values = Arrays.*asList*(1,2,3,4,5,6);

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

@Override

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

System.***out***.println(t);

}

});

values.forEach(i -> {System.***out***.println(i);}); // by lambda

values.forEach(System.***out***::println); // by passing method reference.

values.forEach(i -> *doubleit*(i));

values.forEach(foreachimple :: *doubleit*);

}

**public** **static** **void** doubleit(Integer i ) {

System.***out***.println(i\*2);

}

}

Stream:

* Values.stream().forEach(System.out::println);
* Values.parallelStream().forEach(System.out::println);
* This will create threads to run in parallel, no of threads depends upon the on the no of cores for processor. So we don’t need to create threads explicitly.
* Stream has lot of methods to process data.
* Stream has methods like intermediate (filter(), map(). ) and Terminate (findFirst(), forEach()).
* We cannot reuse the values of stream once used.
* We cannot also use the stream for second time.
* **package** ConsumerInterfacelearn;
* **import** java.util.Arrays;
* **import** java.util.List;
* **import** java.util.function.BinaryOperator;
* **import** java.util.function.Function;
* **import** java.util.stream.Stream;
* **public** **class** streammethods {
* **public** **static** **void** main(String args[]) {
* List<Integer> values = Arrays.*asList*(1,2,3,4,5,6);
* Stream s = values.stream();
* Function<Integer, Integer> f = **new** Function<Integer, Integer>() {
* @Override
* **public** Integer apply(Integer t) {
* **return** t\*2;
* }
* };
* Stream s1 = s.map(f);
* System.***out***.println(s1);
* BinaryOperator<Integer> b = **new** BinaryOperator<Integer>() {
* @Override
* **public** Integer apply(Integer arg0, Integer arg1) {
* **return** arg0+arg1;
* }
* };
* Integer result = (Integer) s1.reduce(0,b);//initial value and binary operator object
* System.***out***.println(result); }
* }
* Map method will take the function interface
* Reduce method will take binary operator interface.
* Filter method will take predicate interface.
* findFirst will return a optional value as output. We can use “orElse” in combination with findFirst.
* Map and filter methods are lazy evaluation methods. They will execute when only they are required.

A screenshot of a cell phone

Description automatically generated