**Java 8**

**1)Lambda Expressions:**

* Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming, and simplifies the development a lot.
* Syntax

**parameter -> expression body**

**Following are the important characteristics of a lambda expression.**

1. Optional type declaration − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
2. Optional parenthesis around parameter − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
3. Optional curly braces − No need to use curly braces in expression body if the body contains a single statement.
4. 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.

**Lambda Expressions Example**

**package** com.java8;

**public** **class** LambdaEXpression {

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

LambdaEXpression tester = **new** LambdaEXpression();

//with type declaration

MathOperation addition = (**int** a, **int** b) -> a + b;

//with out type declaration

MathOperation subtraction = (a, b) -> a - b;

//with return statement along with curly braces

MathOperation multiplication = (**int** a, **int** b) -> { **return** a \* b; };

//without return statement and without curly braces

MathOperation division = (**int** a, **int** b) -> a / b;

System.***out***.println("10 + 5 = " + tester.operate(10, 5, addition));

System.***out***.println("10 - 5 = " + tester.operate(10, 5, subtraction));

System.***out***.println("10 x 5 = " + tester.operate(10, 5, multiplication));

System.***out***.println("10 / 5 = " + tester.operate(10, 5, division));

//without parenthesis

GreetingService greetService1 = message -> System.***out***.println("Hello " + message);

//with parenthesis

GreetingService greetService2 = (message) -> System.***out***.println("Hello " + message);

greetService1.sayMessage("Mahesh");

greetService2.sayMessage("Suresh");

}

**interface** MathOperation {

**int** operation(**int** a, **int** b);

}

**interface** GreetingService {

**void** sayMessage(String message);

}

**private** **int** operate(**int** a, **int** b, MathOperation mathOperation) {

**return** mathOperation.operation(a, b);

}

}

**Following are the important points to be considered in the above example.**

* Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single method only. In the above example, we've used various types of lambda expressions to define the operation method of MathOperation interface. Then we have defined the implementation of sayMessage of GreetingService.
* Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

**Scope**

* Using lambda expression, you can refer to any final variable or effectively final variable (which is assigned only once). Lambda expression throws a compilation error, if a variable is assigned a value the second time.

**Scope Example**

**package** com.java8;

**public** **class** LambdaEXpressionScope {

**final** **static** String ***salutation*** = "Hello! ";

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

GreetingService greetService1 = message -> System.***out***.println(***salutation*** + message);

greetService1.sayMessage("Mahesh");

GreetingService greetService2 = message -> System.***out***.println(***salutation*** + message);

greetService2.sayMessage("shilpa");

}

**interface** GreetingService {

**void** sayMessage(String message);

}

}

**2)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**

**Constructors using new operator (TreeSet::new)**

**Method Reference Example**

**package** com.java8;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** MethodReference {

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

List names = **new** ArrayList();

names.add("Mahesh");

names.add("Suresh");

names.add("Ramesh");

names.add("Naresh");

names.add("Kalpesh");

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

}}

Here we have passed System.out::println method as a static method reference.

**3)Functional Interfaces:**

* Functional interfaces have a single functionality to exhibit.
* For example, a Comparable interface with a single method ‘compareTo’ is used for comparison purpose.
* Java 8 has defined a lot of functional interfaces to be used extensively in lambda expressions.
* Following is the list of functional interfaces defined in java.util.Function package.

**Functional Interface Example**

* Predicate <T> interface is a functional interface with a method test(Object) to return a Boolean value. This interface signifies that an object is tested to be true or false.

**package** com.java8;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.Predicate;

**public** **class** FunctionalInterface {

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

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7, 8, 9);

// Predicate<Integer> predicate = n -> true

// n is passed as parameter to test method of Predicate interface

// test method will always return true no matter what value n has.

System.***out***.println("Print all numbers:");

//pass n as parameter

*eval*(list, n->**true**);

// Predicate<Integer> predicate1 = n -> n%2 == 0

// n is passed as parameter to test method of Predicate interface

// test method will return true if n%2 comes to be zero

System.***out***.println("Print even numbers:");

*eval*(list, n-> n%2 == 0 );

// Predicate<Integer> predicate2 = n -> n > 3

// n is passed as parameter to test method of Predicate interface

// test method will return true if n is greater than 3.

System.***out***.println("Print numbers greater than 3:");

*eval*(list, n-> n > 3 );

}

**public** **static** **void** eval(List<Integer> list, Predicate<Integer> predicate) {

**for**(Integer n: list) {

**if**(predicate.test(n)) {

System.***out***.println(n + " ");

}

}

}

}

Here we've passed Predicate interface, which takes a single input and returns Boolean.

* **Given below is the list of interfaces in Java8.**

1. **BiConsumer<T,U>**Represents an operation that accepts two input arguments, and returns no result.
2. **BiFunction<T,U,R>**Represents a function that accepts two arguments and produces a result.
3. **BinaryOperator<T>**Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
4. **BiPredicate<T,U>**Represents a predicate (Boolean-valued function) of two arguments.
5. **BooleanSupplier**Represents a supplier of Boolean-valued results.
6. **Consumer<T>**Represents an operation that accepts a single input argument and returns no result.
7. **DoubleBinaryOperator**Represents an operation upon two double-valued operands and producing a double-valued result.
8. **DoubleConsumer**Represents an operation that accepts a single double-valued argument and returns no result.
9. **DoubleFunction<R>**Represents a function that accepts a double-valued argument and produces a result.
10. **DoublePredicate**Represents a predicate (Boolean-valued function) of one double-valued argument.
11. **DoubleSupplier**Represents a supplier of double-valued results.
12. **DoubleToIntFunction**Represents a function that accepts a double-valued argument and produces an int-valued result.
13. **DoubleToLongFunction**Represents a function that accepts a double-valued argument and produces a long-valued result.
14. **DoubleUnaryOperator**Represents an operation on a single double-valued operand that produces a double-valued result.
15. **Function<T,R>**Represents a function that accepts one argument and produces a result.
16. **IntBinaryOperator**Represents an operation upon two int-valued operands and produces an int-valued result.
17. **IntConsumer**Represents an operation that accepts a single int-valued argument and returns no result.
18. **IntFunction<R>**Represents a function that accepts an int-valued argument and produces a result.
19. **IntPredicate**Represents a predicate (Boolean-valued function) of one int-valued argument.
20. **IntSupplier**Represents a supplier of int-valued results.
21. **IntToDoubleFunction**Represents a function that accepts an int-valued argument and produces a double-valued result.
22. **IntToLongFunction**Represents a function that accepts an int-valued argument and produces a long-valued result.
23. **IntUnaryOperator**Represents an operation on a single int-valued operand that produces an int-valued result.
24. **LongBinaryOperator**Represents an operation upon two long-valued operands and produces a long-valued result.
25. **LongConsumer**Represents an operation that accepts a single long-valued argument and returns no result.
26. **LongFunction<R>**Represents a function that accepts a long-valued argument and produces a result.
27. **LongPredicate**Represents a predicate (Boolean-valued function) of one long-valued argument.
28. **LongSupplier**Represents a supplier of long-valued results.
29. **LongToDoubleFunction**Represents a function that accepts a long-valued argument and produces a double-valued result.
30. **LongToIntFunction**Represents a function that accepts a long-valued argument and produces an int-valued result.
31. **LongUnaryOperator**Represents an operation on a single long-valued operand that produces a long-valued result.
32. **ObjDoubleConsumer<T>**Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
33. **ObjIntConsumer<T>**Represents an operation that accepts an object-valued and an int-valued argument, and returns no result.
34. **ObjLongConsumer<T>**Represents an operation that accepts an object-valued and a long-valued argument, and returns no result.
35. **Predicate<T>**Represents a predicate (Boolean-valued function) of one argument.
36. **Supplier<T>**Represents a supplier of results.
37. **ToDoubleBiFunction<T,U>**Represents a function that accepts two arguments and produces a double-valued result.
38. **ToDoubleFunction<T>**Represents a function that produces a double-valued result.
39. **ToIntBiFunction<T,U>**Represents a function that accepts two arguments and produces an int-valued result.
40. **ToIntFunction<T>**Represents a function that produces an int-valued result.
41. **ToLongBiFunction<T,U>**Represents a function that accepts two arguments and produces a long-valued result.
42. **ToLongFunction<T>**Represents a function that produces a long-valued result.
43. **UnaryOperator<T>**Represents an operation on a single operand that produces a result of the same type as its operand.

**4)Default Methods:**

* Java 8 introduces a new concept of default method implementation in interfaces. This capability is added for backward compatibility so that old interfaces can be used to leverage the lambda expression capability of Java 8.
* For example, ‘List’ or ‘Collection’ interfaces do not have ‘forEach’ method declaration. Thus, adding such method will simply break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not implement the same.
* Syntax

**public interface vehicle {**

**default void print() {**

**System.out.println("I am a vehicle!");**

**}**

**}**

## Multiple Defaults

* With default functions in interfaces, there is a possibility that a class is implementing two interfaces with same default methods. The following code explains how this ambiguity can be resolved

**public** **interface** vehicle {

**default** **void** print() {

System.***out***.println("I am a vehicle!");

}}

**public** **interface** fourWheeler {

**default** **void** print() {

System.***out***.println("I am a four wheeler!");

}

}

First solution is to create an own method that overrides the default implementation.

**public** **class** car **implements** vehicle, fourWheeler {

**public** **void** print() {

System.***out***.println("I am a four wheeler car vehicle!");

}

}

Second solution is to call the default method of the specified interface using super.

**public** **class** car **implements** vehicle, fourWheeler {

**public** **void** print() {

vehicle.**super**.print();

}

}

**Static Default Methods**

* An interface can also have static helper methods from Java 8 onwards.

**public** **interface** vehicle {

**default** **void** print() {

System.***out***.println("I am a vehicle!");

}

**static** **void** blowHorn() {

System.***out***.println("Blowing horn!!!");

}

}

## Default Method Example

**public** **class** DefaultMethod {

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

Vehicle vehicle = **new** Car();

vehicle.print();

}

}

**interface** Vehicle {

**default** **void** print() {

System.***out***.println("I am a vehicle!");

}

**static** **void** blowHorn() {

System.***out***.println("Blowing horn!!!");

}

}

**interface** FourWheeler {

**default** **void** print() {

System.***out***.println("I am a four wheeler!");

}

}

**class** Car **implements** Vehicle, FourWheeler {

**public** **void** print() {

Vehicle.**super**.print();

FourWheeler.**super**.print();

Vehicle.*blowHorn*();

System.***out***.println("I am a car!");

}

}

**5)Streams:**

* Stream is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way similar to SQL statements. For example, consider the following SQL statement.

**SELECT max(salary), employee\_id, employee\_name FROM Employee**

* The above SQL expression automatically returns the maximum salaried employee's details, without doing any computation on the developer's end. Using collections framework in Java, a developer has to use loops and make repeated checks. Another concern is efficiency; as multi-core processors are available at ease, a Java developer has to write parallel code processing that can be pretty error-prone.
* To resolve such issues, Java 8 introduced the concept of stream that lets the developer to process data declaratively and leverage multicore architecture without the need to write any specific code for it.

**What is Stream?**

* Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

1. **Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
2. **Source** − Stream takes Collections, Arrays, or I/O resources as input source.
3. **Aggregate operations** − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
4. **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
5. **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

**Generating Streams**

With Java 8, Collection interface has two methods to generate a Stream.

1. **stream()** − Returns a sequential stream considering collection as its source.
2. **parallelStream()** − Returns a parallel Stream considering collection as its source.

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

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

**forEach**

* Stream has provided a new method ‘forEach’ to iterate each element of the stream. The following code segment shows how to print 10 random numbers using forEach.

Random random = new Random();

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

**map**

* The ‘map’ method is used to map each element to its corresponding result. The following code segment prints unique squares of numbers using map.

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

//get list of unique squares

List<Integer> squaresList = numbers.stream().map( i -> i\*i).distinct().collect(Collectors.toList());

**filter**

* The ‘filter’ method is used to eliminate elements based on a criteria. The following code segment prints a count of empty strings using filter.

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

//get count of empty string

int count = strings.stream().filter(string -> string.isEmpty()).count();

**limit**

* The ‘limit’ method is used to reduce the size of the stream. The following code segment shows how to print 10 random numbers using limit.

Random random = new Random();

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

**sorted**

* The ‘sorted’ method is used to sort the stream. The following code segment shows how to print 10 random numbers in a sorted order.

Random random = new Random();

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

**Parallel Processing**

* parallelStream is the alternative of stream for parallel processing. Take a look at the following code segment that prints a count of empty strings using parallelStream.

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

//get count of empty string

long count = strings.parallelStream().filter(string -> string.isEmpty()).count();

**It is very easy to switch between sequential and parallel streams.**

**Collectors**

* Collectors are used to combine the result of processing on the elements of a 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);

**Statistics**

* With Java 8, statistics collectors are 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());

## Stream Example

**public** **class** Stream {

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

System.***out***.println("Using Java 7: ");

// Count empty strings

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

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

**long** count = *getCountEmptyStringUsingJava7*(strings);

System.***out***.println("Empty Strings: " + count);

count = *getCountLength3UsingJava7*(strings);

System.***out***.println("Strings of length 3: " + count);

//Eliminate empty string

List<String> filtered = *deleteEmptyStringsUsingJava7*(strings);

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

//Eliminate empty string and join using comma.

String mergedString = *getMergedStringUsingJava7*(strings,", ");

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

List<Integer> numbers = Arrays.*asList*(3, 2, 2, 3, 7, 3, 5);

//get list of square of distinct numbers

List<Integer> squaresList = *getSquares*(numbers);

System.***out***.println("Squares List: " + squaresList);

List<Integer> integers = Arrays.*asList*(1,2,13,4,15,6,17,8,19);

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

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

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

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

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

System.***out***.println("Random Numbers: ");

//print ten random numbers

Random random = **new** Random();

**for**(**int** i = 0; i < 10; i++) {

System.***out***.println(random.nextInt());

}

System.***out***.println("Using Java 8: ");

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

count = strings.stream().filter(string->string.isEmpty()).count();

System.***out***.println("Empty Strings: " + count);

count = strings.stream().filter(string -> string.length() == 3).count();

System.***out***.println("Strings of length 3: " + count);

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

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

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

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

squaresList = numbers.stream().map( i ->i\*i).distinct().collect(Collectors.*toList*());

System.***out***.println("Squares List: " + squaresList);

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

IntSummaryStatistics stats = integers.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());

System.***out***.println("Random Numbers: ");

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

//parallel processing

count = strings.parallelStream().filter(string -> string.isEmpty()).count();

System.***out***.println("Empty Strings: " + count);

}

**private** **static** **int** getCountEmptyStringUsingJava7(List<String> strings) {

**int** count = 0;

**for**(String string: strings) {

**if**(string.isEmpty()) {

count++;

}

}

**return** count;

}

**private** **static** **int** getCountLength3UsingJava7(List<String> strings) {

**int** count = 0;

**for**(String string: strings) {

**if**(string.length() == 3) {

count++;

}

}

**return** count;

}

**private** **static** List<String> deleteEmptyStringsUsingJava7(List<String> strings) {

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

**for**(String string: strings) {

**if**(!string.isEmpty()) {

filteredList.add(string);

}

}

**return** filteredList;

}

**private** **static** String getMergedStringUsingJava7(List<String> strings, String separator) {

StringBuilder stringBuilder = **new** StringBuilder();

**for**(String string: strings) {

**if**(!string.isEmpty()) {

stringBuilder.append(string);

stringBuilder.append(separator);

}

}

String mergedString = stringBuilder.toString();

**return** mergedString.substring(0, mergedString.length()-2);

}

**private** **static** List<Integer> getSquares(List<Integer> numbers) {

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

**for**(Integer number: numbers) {

Integer square = **new** ~~Integer~~(number.intValue() \* number.intValue());

**if**(!squaresList.contains(square)) {

squaresList.add(square);

}

}

**return** squaresList;

}

**private** **static** **int** getMax(List<Integer> numbers) {

**int** max = numbers.get(0);

**for**(**int** i = 1;i < numbers.size();i++) {

Integer number = numbers.get(i);

**if**(number.intValue() > max) {

max = number.intValue();

}

}

**return** max;

}

**private** **static** **int** getMin(List<Integer> numbers) {

**int** min = numbers.get(0);

**for**(**int** i= 1;i < numbers.size();i++) {

Integer number = numbers.get(i);

**if**(number.intValue() < min) {

min = number.intValue();

}

}

**return** min;

}

**private** **static** **int** getSum(List numbers) {

**int** sum = (**int**)(numbers.get(0));

**for**(**int** i = 1;i < numbers.size();i++) {

sum += (**int**)numbers.get(i);

}

**return** sum;

}

**private** **static** **int** getAverage(List<Integer> numbers) {

**return** *getSum*(numbers) / numbers.size();

}

}

**6)Optional Class:**

* Optional is a container object used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values. It is introduced in Java 8 and is similar to what Optional is in Guava.

## Class Declaration-Following is the declaration for java.util.Optional<T> class

**public final class Optional<T> extends Object**

## Class Method

1. **static <T> Optional<T> empty()**Returns an empty Optional instance.
2. **boolean equals(Object obj)**Indicates whether some other object is "equal to" this Optional.
3. **Optional<T> filter(Predicate<? super <T> predicate)**If a value is present and the value matches a given predicate, it returns an Optional describing the value, otherwise returns an empty Optional.
4. **<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)**If a value is present, it applies the provided Optional-bearing mapping function to it, returns that result, otherwise returns an empty Optional.
5. **T get()**If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
6. **int hashCode()**Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.
7. **void ifPresent(Consumer<? super T> consumer)**If a value is present, it invokes the specified consumer with the value, otherwise does nothing.
8. **boolean isPresent()**Returns true if there is a value present, otherwise false.
9. **<U>Optional<U> map(Function<? super T,? extends U> mapper)**If a value is present, applies the provided mapping function to it, and if the result is non-null, returns an Optional describing the result.
10. **static <T> Optional<T> of(T value)**Returns an Optional with the specified present non-null value.
11. **static <T> Optional<T> ofNullable(T value)**Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
12. **T orElse(T other)**Returns the value if present, otherwise returns other.
13. **T orElseGet(Supplier<? extends T> other)**Returns the value if present, otherwise invokes other and returns the result of that invocation.
14. **<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)**Returns the contained value, if present, otherwise throws an exception to be created by the provided supplier.
15. **String toString()**Returns a non-empty string representation of this Optional suitable for debugging.

**This class inherits methods from the following class −**

**java.lang.Object**

**Optional Example**

**public** **class** OptionalExample {

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

OptionalExample java8Tester = **new** OptionalExample();

Integer value1 = **null**;

Integer value2 = **new** ~~Integer~~(10);

//Optional.ofNullable - allows passed parameter to be null.

Optional<Integer> a = Optional.*ofNullable*(value1);

//Optional.of - throws NullPointerException if passed parameter is null

Optional<Integer> b = Optional.*of*(value2);

System.***out***.println(java8Tester.sum(a,b));

}

**public** Integer sum(Optional<Integer> a, Optional<Integer> b) {

//Optional.isPresent - checks the value is present or not

System.***out***.println("First parameter is present: " + a.isPresent());

System.***out***.println("Second parameter is present: " + b.isPresent());

//Optional.orElse - returns the value if present otherwise returns

//the default value passed.

Integer value1 = a.orElse(**new** ~~Integer~~(0));

//Optional.get - gets the value, value should be present

Integer value2 = b.get();

**return** value1 + value2;

}

}

**7)Nashorn Javascript:**

* With Java 8, Nashorn, a much improved javascript engine is introduced, to replace the existing Rhino. Nashorn provides 2 to 10 times better performance, as it directly compiles the code in memory and passes the bytecode to JVM. Nashorn uses invoke dynamics feature, introduced in Java 7 to improve performance.

## jjs

* For Nashorn engine, JAVA 8 introduces a new command line tool, **jjs,** to execute javascript codes at console.

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()

>>

**Pass Arguments**

Open the console and use the following command.

C:\JAVA> jjs -- a b c

jjs> print('letters: ' +arguments.join(", "))

letters: a, b, c

jjs>

**Calling JavaScript from Java**

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

**Example**

**import** javax.script.ScriptEngineManager;

**import** javax.script.ScriptEngine;

**import** javax.script.ScriptException;

**public** **class** NashornJavaScript {

**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());

}

}

**Calling Java from JavaScript**

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

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);

**8)Nashorn Javascript:**