Java 8 - Overview

JAVA 8 is a major feature release of JAVA programming language development. Its initial version was released on 18 March 2014. With the Java 8 release, Java provided supports for functional programming, new JavaScript engine, new APIs for date time manipulation, new streaming API, etc.

New Features

- Lambda Expressions: Adds functional processing capability to Java.
- Method References: Referencing functions by their names instead of invoking them directly. Using functions as parameter.
- Default Methods: Interface to have default method implementation.
- Streams: New stream API to facilitate pipeline processing.
- New Date/Time API: Improved date time API.
- Optional Class: Emphasis on best practices to handle null values properly.
- Nashorn JavaScript: A Java-based engine to execute JavaScript code.

Consider the following code snippet.

Demo1.java

```
 1  package intro;
 2
 3  import java.util.Collections;
 4  import java.util.List;
 5  import java.util.ArrayList;
 6  import java.util.Comparator;
 7
 8  public class Demo1 {
 9    public static void main(String args[]) {
10      List<String> names1 = new ArrayList<>();
11      names1.add("Apple ");
12      names1.add("Orange ");
13      names1.add("Melon ");
14      names1.add("Banana ");
15
16      List<String> names2 = new ArrayList<>();
17      names2.add("Apple ");
18      names2.add("Orange ");
19      names2.add("Melon ");
20      names2.add("Banana ");
21
22      System.out.println("Sort using Java 7 syntax:");
23
24      sortUsingJava7(names1);
25      System.out.println(names1);
26      System.out.println("Sort using Java 8 syntax:");
27
28      sortUsingJava8(names2);
29      System.out.println(names2);
30    }
31    static private void sortUsingJava7(List<String> names) {
32      Collections.sort(names, new Comparator<String>() {
33        @Override
34        public int compare(String s1, String s2) {
35          return s1.compareTo(s2);
36        }
37      });
38    }
39    static private void sortUsingJava8(List<String> names) {
40      Collections.sort(names, (s1, s2) -> s1.compareTo(s2));
41    }
42  }
```

Run the program to get the following result.

```
Sort using Java 7 syntax:
[ Apple Orange Melon Banana ]
Sort using Java 8 syntax:
[ Apple Orange Melon Banana ]
```

Here the **sortUsingJava8()** method uses sort function with a lambda expression as parameter to get the sorting criteria.

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

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.

Lambda Expressions Example

Create the following Java program using any editor of your choice.

Demo1.java

```
 1  package lambda;
 2
 3  public class Demo1 {
 4    public static void main(String args[]) {
 5      //with type declaration
 6      MathOperation addition = (int a, int b) -> a + b;
 7
 8      //with out type declaration
 9      MathOperation subtraction = (a, b) -> a - b;
10
11      //with return statement along with curly braces
12      MathOperation multiplication = (int a, int b) -> { return a * b; };
13
14      //without return statement and without curly braces
15      MathOperation division = (int a, int b) -> a / b;
16      System.out.println("10+5="+operate(10, 5, addition));
17      System.out.println("10-5="+operate(10, 5, subtraction));
18      System.out.println("10x5="+operate(10, 5, multiplication));
19      System.out.println("10/5="+operate(10, 5, division));
20
21      //without parenthesis
```

```
22      GreetingService greetService1 = message ->
23              System.out.println("Hello " + message);
24
25      //with parenthesis
26      GreetingService greetService2 = (message) ->
27              System.out.println("Hello " + message);
28
29      greetService1.sayMessage("Ali");
30      greetService2.sayMessage("Abu");
31    }
32    interface MathOperation {
33      int operation(int a, int b);
34    }
35    interface GreetingService {
36      void sayMessage(String message);
37    }
38    private static int operate(int a, int b, MathOperation mathOperation){
39      return mathOperation.operation(a, b);
40    }
41  }
```

Run the program to get the following result.

```
10+5=15
10-5=5
10x5=50
10/5=2
Hello Ali
Hello Abu
```

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

Create the following Java program using any editor of your choice.

Demo2.java

```
 1  package lambda;
 2
 3  public class Demo2 {
 4    interface GreetingService {
 5        void sayMessage(String message);
 6    }
 7
 8    final static String salutation = "Hello! ";
 9
10    public static void main(String args[]) {
11      GreetingService greetService1 = message ->
12              System.out.println(salutation + message);
13      greetService1.sayMessage("Ali");
14    }
15  }
```

Run the program to get the following result.

```
Hello! Ali
```

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

Create the following Java program using any editor of your choice.

Demo1.java

```
 1  package methodReference;
 2
 3  import java.util.List;
 4  import java.util.ArrayList;
 5
 6  public class Demo1 {
 7
 8    public static void main(String args[]) {
 9      List names = new ArrayList();
10
11      names.add("Apple");
12      names.add("Orange");
13      names.add("Melon");
14      names.add("Bsnana");
15
16      names.forEach(System.out::println);
17    }
18  }
```

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

Run the program to get the following result.

```
Apple
Orange
Melon
Banana
```

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.

Given below is the list of interfaces in Java8.

No.	Interface & Description
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.

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.

Create the following Java program using any editor of your choice.

Demo1.java

```
 1  package functionalInterface;
 2
 3  import java.util.Arrays;
 4  import java.util.List;
 5  import java.util.function.Predicate;
 6
 7  public class Demo1 {
 8
 9    public static void main(String args[]) {
10      List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
11
12      // Predicate<Integer> predicate = n -> true
13      // n is passed as parameter to test method of Predicate interface
14      // test method will always return true no matter what value n has.
15
16      System.out.println("Print all numbers:");
17
18      //pass n as parameter
19      eval(list, n->true);
20
21      // Predicate<Integer> predicate1 = n -> n%2 == 0
22      // n is passed as parameter to test method of Predicate interface
23      // test method will return true if n%2 comes to be zero
24
25      System.out.println("Print even numbers:");
26      eval(list, n-> n%2 == 0 );
27
28      // Predicate<Integer> predicate2 = n -> n > 3
29      // n is passed as parameter to test method of Predicate interface
30      // test method will return true if n is greater than 3.
31
32      System.out.println("Print numbers greater than 3:");
33      eval(list, n-> n > 3 );
34    }
35
36    public static void eval(List<Integer> list,
37                            Predicate<Integer> predicate) {
38      for(Integer n:list) {
39        if(predicate.test(n)) System.out.println(n + " ");
40      }
41    }
42  }
```

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

Run the program to get the following result.

```
Print all numbers:
1
2
3
4
5
6
7
8
9
Print even numbers:
2
4
6
8
Print numbers greater than 3:
4
5
6
7
8
9
```

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

Create the following Java program using any editor of your choice.

Demo1.java

```
 1  package defaultMethods;
 2
 3  public class Demo1 {
 4    public static void main(String args[]) {
 5        Vehicle vehicle = new Car();
 6        vehicle.print();
 7    }
 8  }
 9  interface Vehicle {
10    default void print() {
11        System.out.println("I am a vehicle!");
12    }
13    static void blowHorn() {
14        System.out.println("Blowing horn!!!");
15    }
16  }
17  interface FourWheeler {
18    default void print() {
19        System.out.println("I am a four wheeler!");
20    }
21  }
22  class Car implements Vehicle, FourWheeler {
23    public void print() {
24        Vehicle.super.print();
25        FourWheeler.super.print();
26        Vehicle.blowHorn();
27        System.out.println("I am a car!");
28    }
29  }
```

Run the program to get the following result.

```
I am a vehicle!
I am a four wheeler!
Blowing horn!!!
I am a car!
```

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:

- **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.
- **Source**: Stream takes Collections, Arrays, or I/O resources as input source.
- **Aggregate operations**: Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
- **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.
- **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.

- **stream()**: Returns a sequential stream considering collection as its source.
- **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(s->!s.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(
      s -> s.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(
      s -> !s.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List:" + filtered);
String mergedString = strings.stream().filter(
      s -> !s.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

Create the following Java program using any editor of your choice.

Demo1.java

```
1  package streams;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5  import java.util.IntSummaryStatistics;
6  import java.util.List;
7  import java.util.Random;
8  import java.util.stream.Collectors;
9
10 public class Demo1 {
11   public static void main(String args[]) {
12     System.out.println("Using Java 7:");
13
14     // Count empty strings
15     List<String> strings =
16             Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
17     System.out.println("List:" +strings);
18     long count = getCountEmptyStringUsingJava7(strings);
19
20     System.out.println("Empty Strings:" + count);
21     count = getCountLength3UsingJava7(strings);
22
23     System.out.println("Strings of length 3:" + count);
24
25     //Eliminate empty string
26     List<String> filtered = deleteEmptyStringsUsingJava7(strings);
27     System.out.println("Filtered List:" + filtered);
28
29     //Eliminate empty string and join using comma.
30     String mergedString = getMergedStringUsingJava7(strings,", ");
31     System.out.println("Merged String:" + mergedString);
32     List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
33
34     //get list of square of distinct numbers
35     List<Integer> squaresList = getSquares(numbers);
36     System.out.println("Squares List:" + squaresList);
37     List<Integer> integers = Arrays.asList(1,2,13,4,15,6,17,8,19);
38
39     System.out.println("List:"+integers);
40     System.out.println("Highest number in List:"+getMax(integers));
41     System.out.println("Lowest number in List:"+getMin(integers));
42     System.out.println("Sum of all numbers:"+getSum(integers));
43     System.out.println("Average of all numbers:"+getAverage(integers));
44     System.out.println("Random Numbers:");
45
46     //print ten random numbers
47     Random random = new Random();
48
49     for(int i = 0; i < 10; i++) System.out.println(random.nextInt());
50
```

```
51      System.out.println("Using Java 8:");
52      System.out.println("List:" +strings);
53
54      count = strings.stream().filter(s->s.isEmpty()).count();
55      System.out.println("Empty Strings:" + count);
56
57      count = strings.stream().filter(s -> s.length() == 3).count();
58      System.out.println("Strings of length 3:" + count);
59
60      filtered = strings.stream().filter(
61              s ->!s.isEmpty()).collect(Collectors.toList());
62      System.out.println("Filtered List:" + filtered);
63
64      mergedString = strings.stream().filter(
65              s ->!s.isEmpty()).collect(Collectors.joining(", "));
66      System.out.println("Merged String:" + mergedString);
67
68      squaresList = numbers.stream().map(
69              i ->i*i).distinct().collect(Collectors.toList());
70      System.out.println("Squares List:" + squaresList);
71      System.out.println("List:" +integers);
72
73      IntSummaryStatistics stats = integers.stream().mapToInt(
74              (x) ->x).summaryStatistics();
75
76      System.out.println("Highest number in List:"+stats.getMax());
77      System.out.println("Lowest number in List:"+stats.getMin());
78      System.out.println("Sum of all numbers:"+stats.getSum());
79      System.out.println("Average of all numbers:"+stats.getAverage());
80      System.out.println("Random Numbers:");
81
82      random.ints().limit(10).sorted().forEach(System.out::println);
83
84      //parallel processing
85      count = strings.parallelStream().filter(s -> s.isEmpty()).count();
86      System.out.println("Empty Strings:" + count);
87    }
88    private static int getCountEmptyStringUsingJava7(List<String>strings){
89      int count = 0;
90
91      for(String string:strings) {
92        if(string.isEmpty()) count++;
93      }
94      return count;
95    }
96    private static int getCountLength3UsingJava7(List<String> strings) {
97      int count = 0;
98
99      for(String string:strings) {
100       if(string.length() == 3) count++;
101     }
102     return count;
103   }
104   private static List<String> deleteEmptyStringsUsingJava7(
105       List<String> strings) {
106     List<String> filteredList = new ArrayList<String>();
107
```

```
108       for(String string:strings) {
109         if(!string.isEmpty()) filteredList.add(string);
110       }
111       return filteredList;
112     }
113     private static String getMergedStringUsingJava7(
114         List<String> strings, String separator) {
115       StringBuilder stringBuilder = new StringBuilder();
116
117       for(String string:strings) {
118         if(!string.isEmpty()) {
119           stringBuilder.append(string);
120           stringBuilder.append(separator);
121         }
122       }
123       String mergedString = stringBuilder.toString();
124       return mergedString.substring(0, mergedString.length()-2);
125     }
126     private static List<Integer> getSquares(List<Integer> numbers) {
127       List<Integer> squaresList = new ArrayList<Integer>();
128
129       for(Integer number:numbers) {
130         Integer square = new Integer(number.intValue()*number.intValue());
131         if(!squaresList.contains(square)) squaresList.add(square);
132       }
133       return squaresList;
134     }
135     private static int getMax(List<Integer> numbers) {
136       int max = numbers.get(0);
137
138       for(int i = 1;i < numbers.size();i++) {
139         Integer number = numbers.get(i);
140         if(number.intValue() > max) max = number.intValue();
141       }
142       return max;
143     }
144     private static int getMin(List<Integer> numbers) {
145       int min = numbers.get(0);
146
147       for(int i= 1;i < numbers.size();i++) {
148         Integer number = numbers.get(i);
149
150         if(number.intValue() < min) min = number.intValue();
151       }
152       return min;
153     }
154     private static int getSum(List numbers) {
155       int sum = (int)(numbers.get(0));
156
157       for(int i = 1;i < numbers.size();i++) sum += (int)numbers.get(i);
158       return sum;
159     }
160     private static int getAverage(List<Integer> numbers) {
161       return getSum(numbers) / numbers.size();
162     }
164 }
```

Run the program to get the following result.

```
Using Java 7:
List:[abc, , bc, efg, abcd, , jkl]
Empty Strings:2
Strings of length 3:3
Filtered List:[abc, bc, efg, abcd, jkl]
Merged String:abc, bc, efg, abcd, jkl
Squares List:[9, 4, 49, 25]
List:[1, 2, 13, 4, 15, 6, 17, 8, 19]
Highest number in List:19
Lowest number in List:1
Sum of all numbers:85
Average of all numbers:9
Random Numbers:
-1279735475
903418352
-1133928044
-1571118911
628530462
18407523
-881538250
-718932165
270259229
421676854
Using Java 8:
List:[abc, , bc, efg, abcd, , jkl]
Empty Strings:2
Strings of length 3:3
Filtered List:[abc, bc, efg, abcd, jkl]
Merged String:abc, bc, efg, abcd, jkl
Squares List:[9, 4, 49, 25]
List:[1, 2, 13, 4, 15, 6, 17, 8, 19]
Highest number in List:19
Lowest number in List:1
Sum of all numbers:85
Average of all numbers:9.444444444444445
Random Numbers:
-1009474951
-551240647
-2484714
181614550
933444268
1227850416
1579250773
1627454872
1683033687
1798939493
Empty Strings:2
```

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

No.	Method & Description
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

Create the following Java program using any editor of your choice.

Demo1.java

```
1  package optionalClass;
2
3  import java.util.Optional;
4
5  public class Demo1 {
6
7    public static void main(String args[]) {
8      Integer value1 = null;
9      Integer value2 = new Integer(10);
10
11     //Optional.ofNullable - allows passed parameter to be null.
12     Optional<Integer> a = Optional.ofNullable(value1);
13
14     //Optional.of-throws NullPointerException if parameter is null
15     Optional<Integer> b = Optional.of(value2);
16     System.out.println(sum(a,b));
17   }
18
19   static public Integer sum(Optional<Integer> a, Optional<Integer> b) {
20     //Optional.isPresent - checks the value is present or not
21
22     System.out.println("First parameter is present:"+a.isPresent());
23     System.out.println("Second parameter is present:"+b.isPresent());
24
25     //Optional.orElse - returns the value if present otherwise returns
26     //  the default value passed.
27     Integer value1 = a.orElse(new Integer(0));
28
29     //Optional.get - gets the value, value should be present
30     Integer value2 = b.get();
31     return value1 + value2;
32   }
33 }
```

Run the program to get the following result.

```
First parameter is present:false
Second parameter is present:true
10
```

New Date/Time API

With Java 8, a new Date-Time API is introduced to cover the following drawbacks of old date-time API.

- **Not thread safe**: java.util.Date is not thread safe, thus developers have to deal with concurrency issue while using date. The new date-time API is immutable and does not have setter methods.
- **Poor design**: Default Date starts from 1900, month starts from 1, and day starts from 0, so no uniformity. The old API had less direct methods for date operations. The new API provides numerous utility methods for such operations.
- **Difficult time zone handling**: Developers had to write a lot of code to deal with timezone issues. The new API has been developed keeping domain-specific design in mind.

Java 8 introduces a new date-time API under the package java.time. Following are some of the important classes introduced in java.time package.

- **Local**: Simplified date-time API with no complexity of timezone handling.
- **Zoned**: Specialized date-time API to deal with various timezones.

Local Date-Time API

LocalDate/LocalTime and LocalDateTime classes simplify the development where timezones are not required. Let's see them in action.

Create the following java program using any editor of your choice.

Java8Tester.java

```
 1  package dateTimeAPI;
 2
 3  import java.time.LocalDate;
 4  import java.time.LocalTime;
 5  import java.time.LocalDateTime;
 6  import java.time.Month;
 7
 8  public class Demo1 {
 9    public static void main(String args[]) {
10      testLocalDateTime();
11    }
12    static public void testLocalDateTime() {
13      LocalDateTime currentTime = LocalDateTime.now();
14      System.out.println("Current DateTime:" + currentTime);
15      LocalDate date1 = currentTime.toLocalDate();
16      System.out.println("date1:" + date1);
17      Month month = currentTime.getMonth();
18      int day = currentTime.getDayOfMonth();
19      int seconds = currentTime.getSecond();
20      System.out.println("Month:"+month+" day:"+day+" seconds:"+seconds);
21      LocalDateTime date2=currentTime.withDayOfMonth(10).withYear(2012);
22      System.out.println("date2:" + date2);
23
```

```
24      //12 december 2014
25      LocalDate date3 = LocalDate.of(2014, Month.DECEMBER, 12);
26      System.out.println("date3:" + date3);
27
28      LocalTime date4 = LocalTime.of(22, 15); //22 hour 15 minutes
29      System.out.println("date4:" + date4);
30
31      LocalTime date5 = LocalTime.parse("20:15:30");//parse a string
32      System.out.println("date5:" + date5);
33    }
34  }
```

Run the program to get the following result.

```
Current DateTime:2014-12-09T11:00:45.457
date1:2014-12-09
Month:DECEMBER day:9 seconds:45
date2:2012-12-10T11:00:45.457
date3:2014-12-12
date4:22:15
date5:20:15:30
```

Zoned Date-Time API

Zoned date-time API is to be used when time zone is to be considered. Let us see them in action.

Create the following Java program using any editor of your choice.

Demo2.java

```
1   package dateTimeAPI;
2
3   import java.time.ZonedDateTime;
4   import java.time.ZoneId;
5
6   public class Demo2 {
7     public static void main(String args[]) {
8       testZonedDateTime();
9     }
10    static public void testZonedDateTime() {
11      // Get the current date and time
12      ZonedDateTime date1 = ZonedDateTime.parse(
13        "2007-12-03T10:15:30+05:30[Asia/Karachi]");
14      System.out.println("date1:" + date1);
15      ZoneId id = ZoneId.of("Europe/Paris");
16      System.out.println("ZoneId:" + id);
17      ZoneId currentZone = ZoneId.systemDefault();
18      System.out.println("CurrentZone:" + currentZone);
19    }
20  }
```

Run the program to get the following result.

```
date1:2007-12-03T10:15:30+05:00[Asia/Karachi]
ZoneId:Europe/Paris
CurrentZone:Etc/UTC
```

Chrono Units Enum

java.time.temporal.ChronoUnit enum is added in Java 8 to replace the integer values used in old API to represent day, month, etc. Let us see them in action.

Create the following Java program using any editor of your choice.

Demo3.java

```
1  package dateTimeAPI;
2
3  import java.time.LocalDate;
4  import java.time.temporal.ChronoUnit;
5
6  public class Demo3 {
7    public static void main(String args[]) {
8      testChromoUnits();
9    }
10
11   static public void testChromoUnits() {
12     //Get the current date
13     LocalDate today = LocalDate.now();
14     System.out.println("Current date:" + today);
15
16     //add 1 week to the current date
17     System.out.println("Next week:"+today.plus(1,ChronoUnit.WEEKS));
18
19     //add 1 month to the current date
20     System.out.println("Next month:"+today.plus(1,ChronoUnit.MONTHS));
21
22     //add 1 year to the current date
23     System.out.println("Next year:"+today.plus(1,ChronoUnit.YEARS));
24
25     //add 10 years to the current date
26     LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);
27     System.out.println("Date after ten year:" + nextDecade);
28   }
29 }
```

Run the program to get the following result.

```
Current date:2014-12-10
Next week:2014-12-17
Next month:2015-01-10
Next year:2015-12-10
Date after ten year:2024-12-10
```

Period and Duration

With Java 8, two specialized classes are introduced to deal with the time differences.

- **Period**: It deals with date based amount of time.
- **Duration**: It deals with time based amount of time.

Let us see them in action.

Create the following Java program using any editor of your choice.

Demo4.java

```
 1  package dateTimeAPI;
 2
 3  import java.time.temporal.ChronoUnit;
 4  import java.time.LocalDate;
 5  import java.time.LocalTime;
 6  import java.time.Duration;
 7  import java.time.Period;
 8
 9  public class Demo4 {
10    public static void main(String args[]) {
11      testPeriod();
12      testDuration();
13    }
14    static public void testPeriod() {
15      //Get the current date
16      LocalDate date1 = LocalDate.now();
17      System.out.println("Current date:" + date1);
18
19      //add 1 month to the current date
20      LocalDate date2 = date1.plus(1, ChronoUnit.MONTHS);
21      System.out.println("Next month:" + date2);
22
23      Period period = Period.between(date2, date1);
24      System.out.println("Period:" + period);
25    }
26    static public void testDuration() {
27      LocalTime time1 = LocalTime.now();
28      Duration twoHours = Duration.ofHours(2);
29
30      LocalTime time2 = time1.plus(twoHours);
31      Duration duration = Duration.between(time1, time2);
32
33      System.out.println("Duration:" + duration);
34    }
35  }
```

Run the program to get the following result.

```
Current date:2014-12-10
Next month:2015-01-10
Period:P-1M
Duration:PT2H
```

Temporal Adjusters

TemporalAdjuster is used to perform the date mathematics. For example, get the "Second Saturday of the Month" or "Next Tuesday". Let us see them in action.

Create the following Java program using any editor of your choice.

Demo5.java

```
 1  package dateTimeAPI;
 2
 3  import java.time.LocalDate;
 4  import java.time.temporal.TemporalAdjusters;
 5  import java.time.DayOfWeek;
 6
 7  public class Demo5{
 8    public static void main(String args[]) {
 9      testAdjusters();
10    }
11    static public void testAdjusters() {
12      //Get the current date
13      LocalDate date1 = LocalDate.now();
14      System.out.println("Current date:" + date1);
15
16      //get the next tuesday
17      LocalDate nextTuesday =
18        date1.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
19      System.out.println("Next Tuesday on:" + nextTuesday);
20
21      //get the second saturday of next month
22      LocalDate firstInYear =
23        LocalDate.of(date1.getYear(),date1.getMonth(), 1);
24      LocalDate secondSaturday = firstInYear.with(TemporalAdjusters.nextOrSame(
25
26      DayOfWeek.SATURDAY)).with(
27        TemporalAdjusters.next(DayOfWeek.SATURDAY));
28      System.out.println("Second Saturday on:" + secondSaturday);
29    }
30  }
```

Run the program to get the following result.

```
Current date:014-12-10
Next Tuesday on:2014-12-16
Second Saturday on:2014-12-13
```

Backward Compatibility

A toInstant() method is added to the original Date and Calendar objects, which can be used to convert them to the new Date-Time API. Use an ofInstant(Insant,ZoneId) method to get a LocalDateTime or ZonedDateTime object. Let us see them in action.

Create the following Java program using any editor of your choice.

Demo6.java

```
 1  package dateTimeAPI;
 2
 3  import java.time.LocalDateTime;
 4  import java.time.ZonedDateTime;
 5  import java.util.Date;
 6  import java.time.Instant;
 7  import java.time.ZoneId;
 8
 9  public class Demo6{
10    public static void main(String args[]) {
11      testBackwardCompatability();
12    }
13    static public void testBackwardCompatability() {
14      //Get the current date
15      Date currentDate = new Date();
16      System.out.println("Current date:" + currentDate);
17
18      //Get the instant of current date in terms of milliseconds
19      Instant now = currentDate.toInstant();
20      ZoneId currentZone = ZoneId.systemDefault();
21
22      LocalDateTime localDateTime =
23              LocalDateTime.ofInstant(now, currentZone);
24      System.out.println("Local date:" + localDateTime);
25
26      ZonedDateTime zonedDateTime =
27              ZonedDateTime.ofInstant(now, currentZone);
28      System.out.println("Zoned date:" + zonedDateTime);
29    }
30  }
```

Run the program to get the following result.

```
Current date:Wed Dec 10 05:44:06 UTC 2014
Local date:2014-12-10T05:44:06.635
Zoned date:2014-12-10T05:44:06.635Z[Etc/UTC]
```

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.

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()
>>
```

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.

Create the following Java program using any editor of your choice.

Demo1.java

```
1  package nashornJS;
2  import javax.script.ScriptEngineManager;
3  import javax.script.ScriptEngine;
4  import javax.script.ScriptException;
5  public class Demo1{
6    public static void main(String args[]) {
7      ScriptEngineManager scriptEngineManager=
8        new ScriptEngineManager();
9      ScriptEngine nashorn =
10       scriptEngineManager.getEngineByName("nashorn");
11     String name = "Ali";
12     Integer result = null;
13     try {
14       nashorn.eval("print('" + name + "')");
15       result = (Integer) nashorn.eval("10 + 2");
16     } catch(ScriptException e) {
17       System.out.println("Error executing script:"+ e.getMessage());
18     }
19     System.out.println(result.toString());
20   }
21 }
```

Run the program to get the following result.

```
Ali
12
```

Calling Java from JavaScript

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

sample.js

```
1  var BigDecimal = Java.type('java.math.BigDecimal');
2  function calculate(amount, percentage) {
3     var result = new BigDecimal(amount).multiply(
4        new BigDecimal(percentage)).divide(
5              new BigDecimal("100"), 2, BigDecimal.ROUND_HALF_EVEN);
6    return result.toPlainString();
7  }
8  var result = calculate(568000000000000000023,13.9);
9  print(result);
```

Open the console and use the following command: `C:\JAVA>jjs sample.js`

It should produce the following output:

```
78952000000000000003.20
```

Base64

With Java 8, Base64 has finally got its due. Java 8 now has inbuilt encoder and decoder for Base64 encoding. In Java 8, we can use three types of Base64 encoding.

- **Simple**: Output is mapped to a set of characters lying in A-Za-z0-9+/. The encoder does not add any line feed in output, and the decoder rejects any character other than A-Za-z0-9+/.
- **URL**: Output is mapped to set of characters lying in A-Za-z0-9+_. Output is URL and filename safe.
- **MIME**: Output is mapped to MIME friendly format. Output is represented in lines of no more than 76 characters each, and uses a carriage return '\r' followed by a linefeed '\n' as the line separator. No line separator is present to the end of the encoded output.

Nested Classes

No.	Nested class & Description
1	**static class Base64.Decoder** This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.
2	**static class Base64.Encoder** This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Methods

No.	Method Name & Description
1	**static Base64.Decoder getDecoder()** Returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.
2	**static Base64.Encoder getEncoder()** Returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.

3	**static Base64.Decoder getMimeDecoder()** Returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.
4	**static Base64.Encoder getMimeEncoder()** Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.
5	**static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)** Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators.
6	**static Base64.Decoder getUrlDecoder()** Returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme.
7	**static Base64.Encoder getUrlEncoder()** Returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme.

Methods Inherited

This class inherits methods from the following class:

- java.lang.Object

Base64 Example

Create the following Java program using any editor of your choice.

Demo1.java

```
1  package base64;
2
3  import java.util.Base64;
4  import java.util.UUID;
5  import java.io.UnsupportedEncodingException;
6
7  public class Demo1 {
8    public static void main(String args[]) {
9      try {
10
11       // Encode using basic encoder
```

```
12          String base64encodedString = Base64.getEncoder().encodeToString(
13                "TutorialsPoint?java8".getBytes("utf-8"));
14          System.out.println("Base64 Encoded String (Basic) " +
15                base64encodedString);
16
17          // Decode
18          byte[] base64decodedBytes =
19                Base64.getDecoder().decode(base64encodedString);
20
21          System.out.println("Original String:" +
22                new String(base64decodedBytes, "utf-8"));
23          base64encodedString = Base64.getUrlEncoder().encodeToString(
24                "TutorialsPoint?java8".getBytes("utf-8"));
25          System.out.println("Base64 Encoded String (URL) " +
26                base64encodedString);
27
28          StringBuilder stringBuilder = new StringBuilder();
29
30          for (int i = 0; i < 10; ++i) {
31            stringBuilder.append(UUID.randomUUID().toString());
32          }
33
34          byte[] mimeBytes = stringBuilder.toString().getBytes("utf-8");
35          String mimeEncodedString =
36                Base64.getMimeEncoder().encodeToString(mimeBytes);
37          System.out.println("Base64 Encoded String (MIME) " +
38                mimeEncodedString);
39
40       } catch(UnsupportedEncodingException e) {
41          System.out.println("Error " + e.getMessage());
42       }
43    }
44 }
```

Run the program to get the following result.

```
Base64 Encoded String (Basic):VHV0b3JpYWxzUG9pbnQ/amF2YTg=
Original String:TutorialsPoint?java8
Base64 Encoded String (URL):VHV0b3JpYWxzUG9pbnQ_amF2YTg=
Base64                          Encoded                          String
(MIME):YmU3NWY2ODktNGM5YS00ODlmLWI2MTUtZTVkOTk2YzQ1Njk1Y2EwZTg2OTEt
MmRiZC00YTQ1LWJl
NTctMTI1MWUwMTk0ZWQyNDE0NDAwYjgtYTYxOS00NDY5LTllYTctNjc1YzE3YWJhZTk
1MTQ2MDQz
NDItOTAyOC00ZWI0LThlOTYtZWU5YzcwNWQyYzVhMTQxMWRjYTMtY2MwNi00MzU0LTg
0MTgtNGQ1
MDkwYjdiMzg2ZTY0OWU5MmUtZmNkYS00YWEwLTg0MjQtYThiOTQxNDQ2YzhhNTVhYWE
xZjItNjU2
Mi00YmM4LTk2ZGYtMDE4YmY5ZDZhMjkwMzM3MWUzNDMtMmQ3MS00MDczLWI0Y2UtMTQ
xODE0MGU5
YjdmYTVlODUxYzItN2NmOS00N2UyLWIyODQtMThlMWVkYTY4M2Q1YjE3YTMyYmItZjl
lMS00MTFk
LWJiM2UtM2JhYzUxYzI5OWI4
```