

## 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 expression** – Adds functional processing capability to Java.
- **Method references** – Referencing functions by their names instead of invoking them directly. Using functions as parameter.
- **Default method** – Interface to have default method implementation.
- **New tools** – New compiler tools and utilities are added like 'jdeps' to figure out dependencies.
- **Stream API** – New stream API to facilitate pipeline processing.
- **Date Time API** – Improved date time API.
- **Optional** – Emphasis on best practices to handle null values properly.
- **Nashorn, JavaScript Engine** – A Java-based engine to execute JavaScript code.

Consider the following code snippet.

```
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;

public class Java8Tester {
    public static void main(String args[]){
        List<String> names1 = new ArrayList<String>();
        names1.add("Mahesh ");
        names1.add("Suresh ");
        names1.add("Ramesh ");
        names1.add("Naresh ");
        names1.add("Kalpesh ");

        List<String> names2 = new ArrayList<String>();
        names2.add("Mahesh ");
        names2.add("Suresh ");
        names2.add("Ramesh ");
        names2.add("Naresh ");
        names2.add("Kalpesh ");

        Java8Tester tester = new Java8Tester();
        System.out.println("Sort using Java 7 syntax: ");

        tester.sortUsingJava7(names1);
        System.out.println(names1);
        System.out.println("Sort using Java 8 syntax: ");

        tester.sortUsingJava8(names2);
        System.out.println(names2);
    }

    private void sortUsingJava7(List<String> names){
        //sort using java 7
        Collections.sort(names, new Comparator<String>() {
```

```

    @Override
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
};

private void sortUsingJava8(List<String> names){
    //sort using java 8
    Collections.sort(names, (s1, s2) -> s1.compareTo(s2));
}
}

```

Run the program to get the following result.

```

Sort using Java 7 syntax:
[ Kalpesh Mahesh Naresh Ramesh Suresh ]
Sort using Java 8 syntax:
[ Kalpesh Mahesh Naresh Ramesh Suresh ]

```

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

## JAVA 8 - ENVIRONMENT SETUP

### Try it OPTion Online

We have set up the Java Programming environment online, so that you can compile and execute all the available examples online. It gives you confidence in what you are reading and enables you to verify the programs with different options. Feel free to modify any example and execute it online.

Try the following example using our online compiler available at

```

public class MyFirstJavaProgram {
    public static void main(String []args) {
        System.out.println("Hello World");
    }
}

```

For most of the examples given in this tutorial, you will find a **Try it** option in our website code sections at the top right corner that will take you to the online compiler. So just make use of it and enjoy your learning.

### Local Environment Setup

If you want to set up your own environment for Java programming language, then this section guides you through the whole process. Please follow the steps given below to set up your Java environment.

Java SE can be downloaded for free from the following link – [Download Java](#).

You download a version based on your operating system.

Follow the instructions to download Java, and run the **.exe** to install Java on your machine. Once you have installed Java on your machine, you would need to set environment variables to point to correct installation directories.

### Setting Up the Path for Windows 2000/XP

Assuming you have installed Java in c:\Program Files\java\jdk directory:

- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.

- Now, alter the 'Path' variable so that it also contains the path to the Java executable. For example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

## Setting Up the Path for Windows 95/98/ME

Assuming you have installed Java in c:\Program Files\java\jdk directory –

- Edit the 'C:\autoexec.bat' file and add the following line at the end – 'SET PATH=%PATH%;C:\Program Files\java\jdk\bin'

## Setting Up the Path for Linux, UNIX, Solaris, FreeBSD

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

For example, if you use bash as your shell, then you would add the following line at the end of your '.bashrc: export PATH=/path/to/java:\$PATH'

## Popular Java Editors

To write Java programs, you need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following –

- **Notepad** – On Windows machine, you can use any simple text editor like Notepad (recommended for this tutorial) or TextPad.
- **Netbeans** – It is a Java IDE that is open-source and free which can be downloaded from <http://www.netbeans.org/index.html>.
- **Eclipse** – It is also a Java IDE developed by the Eclipse open-source community and can be downloaded from <http://www.eclipse.org/>.

# JAVA 8 - 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 infer 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 in, say, C:\> JAVA.

### Java8Tester.java

```

public class Java8Tester {
    public static void main(String args[]){
        Java8Tester tester = new Java8Tester();

        //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));

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

        //without 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);
    }
}

```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```

10 + 5 = 15
10 - 5 = 5
10 x 5 = 50
10 / 5 = 2
Hello Mahesh
Hello Suresh

```

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 in, say, C:\> JAVA.

### Java8Tester.java

```
public class Java8Tester {
    final static String salutation = "Hello! ";
    public static void main(String args[]){
        GreetingService greetService1 = message ->
        System.out.println(salutation + message);
        greetService1.sayMessage("Mahesh");
    }

    interface GreetingService {
        void sayMessage(String message);
    }
}
```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```
Hello! Mahesh
```

## JAVA 8 - 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 in, say, C:\> JAVA.

## Java8Tester.java

```
import java.util.List;
import java.util.ArrayList;

public class Java8Tester {
    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.

### Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```
Mahesh
Suresh
Ramesh
Naresh
Kalpesh
```

## JAVA 8 - 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.

S. No.	Interface & Description
1	<b>BiConsumer&lt;T,U&gt;</b> Represents an operation that accepts two input arguments, and returns no result.
2	<b>BiFunction&lt;T,U,R&gt;</b> Represents a function that accepts two arguments and produces a result.
3	<b>BinaryOperator&lt;T&gt;</b> 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 in, say, C:\> JAVA.

### Java8Tester.java

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class Java8Tester {
    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.

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```
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
```

## JAVA 8 - 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 {
    default 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 {
    default 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 in, say, C:\> JAVA.

### Java8Tester.java

```
public class Java8Tester {
    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!");
    }
}
```

```
}  
}
```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

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

## JAVA 8 - 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(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  
int 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<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

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());
```

## Stream Example

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

### Java8Tester.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.Random;
import java.util.stream.Collectors;
import java.util.Map;

public class Java8Tester {
    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();
}
}

```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce 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 : 9.4444444444444445
Random Numbers:
-1009474951
-551240647
-2484714
181614550
933444268
1227850416
1579250773
1627454872
1683033687
1798939493
Empty Strings: 2
```

## JAVA 8 - 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

S. No.	Method & Description
1	<b>static &lt;T&gt; Optional&lt;T&gt; empty()</b> Returns an empty Optional instance.
2	<b>boolean equals(Object obj)</b> Indicates whether some other object is "equal to" this Optional.
3	<b>Optional&lt;T&gt; filter(Predicate&lt;? super &lt;T&gt; predicate)</b> 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	<b>&lt;U&gt; Optional&lt;U&gt; flatMap(Function&lt;? super T,Optional&lt;U&gt;&gt; mapper)</b> If a value is present, it applies the provided Optional-bearing mapping function to it, returns that result, otherwise returns an empty Optional.
5	<b>T get()</b> If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
6	<b>int hashCode()</b> 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 in, say, C:\> JAVA.

### Java8Tester.java

```
import java.util.Optional;

public class Java8Tester {
    public static void main(String args[]){
        Java8Tester java8Tester = new Java8Tester();
        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;
}
}

```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```

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

```

## JAVA 8 - 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.

```
$jjs sample.js
```

It will produce the following output:

```
Hello World!
```

## jjs in Interactive Mode

Open the console and use the following command.

```
$jjs
jjs> print("Hello, World!")
Hello, World!
jjs> quit()
>>
```

## Pass Arguments

Open the console and use the following command.

```
$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

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

### Java8Tester.java

```
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 –

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$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.

```
$jjs sample.js
```

It should produce the following output –

```
7895200000000000000003.20
```

## JAVA 8 - 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 Data-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 in, say, C:\> JAVA.

## Java8Tester.java

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Month;

public class Java8Tester {
    public static void main(String args[]){
        Java8Tester java8tester = new Java8Tester();
        java8tester.testLocalDateTime();
    }

    public void testLocalDateTime(){

        // Get the current date and time
        LocalDateTime currentTime = LocalDateTime.now();
        System.out.println("Current DateTime: " + currentTime);

        LocalDate date1 = currentTime.toLocalDate();
        System.out.println("date1: " + date1);

        Month month = currentTime.getMonth();
        int day = currentTime.getDayOfMonth();
        int seconds = currentTime.getSecond();

        System.out.println("Month: " + month +"day: " + day +"seconds: " + seconds);

        LocalDateTime date2 = currentTime.withDayOfMonth(10).withYear(2012);
        System.out.println("date2: " + date2);

        //12 december 2014
        LocalDate date3 = LocalDate.of(2014, Month.DECEMBER, 12);
        System.out.println("date3: " + date3);

        //22 hour 15 minutes
        LocalTime date4 = LocalTime.of(22, 15);
        System.out.println("date4: " + date4);

        //parse a string
        LocalTime date5 = LocalTime.parse("20:15:30");
        System.out.println("date5: " + date5);
    }
}
```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```
Current DateTime: 2014-12-09T11:00:45.457
date1: 2014-12-09
Month: DECEMBERday: 9seconds: 45
date2: 2012-12-10T11:00:45.457
date3: 2014-12-12
date4: 22:15
```



date5: 20:15:30

## Zoned Data-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 in, say, C:\> JAVA.

### Java8Tester.java

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class Java8Tester {
    public static void main(String args[]){
        Java8Tester java8tester = new Java8Tester();
        java8tester.testZonedDateTime();
    }

    public void testZonedDateTime(){

        // Get the current date and time
        ZonedDateTime date1 = ZonedDateTime.parse("2007-12-
03T10:15:30+05:30[Asia/Karachi]");
        System.out.println("date1: " + date1);

        ZoneId id = ZoneId.of("Europe/Paris");
        System.out.println("ZoneId: " + id);

        ZoneId currentZone = ZoneId.systemDefault();
        System.out.println("CurrentZone: " + currentZone);
    }
}
```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```
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 in, say, C:\> JAVA.

### Java8Tester.java

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Java8Tester {
    public static void main(String args[]){
```

```

    Java8Tester java8tester = new Java8Tester();
    java8tester.testChromoUnits();
}

public void testChromoUnits(){

    //Get the current date
    LocalDate today = LocalDate.now();
    System.out.println("Current date: " + today);

    //add 1 week to the current date
    LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
    System.out.println("Next week: " + nextWeek);

    //add 1 month to the current date
    LocalDate nextMonth = today.plus(1, ChronoUnit.MONTHS);
    System.out.println("Next month: " + nextMonth);

    //add 1 year to the current date
    LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
    System.out.println("Next year: " + nextYear);

    //add 10 years to the current date
    LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);
    System.out.println("Date after ten year: " + nextDecade);
}
}

```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce 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 & 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 in, say, C:\> JAVA.

### Java8Tester.java

```

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.time.Duration;
import java.time.Period;

```

```

public class Java8Tester {
    public static void main(String args[]){
        Java8Tester java8tester = new Java8Tester();
        java8tester.testPeriod();
        java8tester.testDuration();
    }

    public void testPeriod(){

        //Get the current date
        LocalDate date1 = LocalDate.now();
        System.out.println("Current date: " + date1);

        //add 1 month to the current date
        LocalDate date2 = date1.plus(1, ChronoUnit.MONTHS);
        System.out.println("Next month: " + date2);
        Period period = Period.between(date2, date1);
        System.out.println("Period: " + period);
    }

    public void testDuration(){
        LocalTime time1 = LocalTime.now();
        Duration twoHours = Duration.ofHours(2);

        LocalTime time2 = time1.plus(twoHours);
        Duration duration = Duration.between(time1, time2);

        System.out.println("Duration: " + duration);
    }
}

```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```

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 in, say, C:\> JAVA.

### Java8Tester.java

```

import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
import java.time.DayOfWeek;

public class Java8Tester {
    public static void main(String args[]){
        Java8Tester java8tester = new Java8Tester();

```

```

        java8tester.testAdjusters();
    }

    public void testAdjusters(){

        //Get the current date
        LocalDate date1 = LocalDate.now();
        System.out.println("Current date: " + date1);

        //get the next tuesday
        LocalDate nextTuesday = date1.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
        System.out.println("Next Tuesday on : " + nextTuesday);

        //get the second saturday of next month
        LocalDate firstInYear = LocalDate.of(date1.getYear(), date1.getMonth(), 1);
        LocalDate secondSaturday =
firstInYear.with(TemporalAdjusters.nextOrSame(DayOfWeek.SATURDAY)).with(TemporalAdjusters.
next(DayOfWeek.SATURDAY));
        System.out.println("Second Saturday on : " + secondSaturday);
    }
}

```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following result –

```

Current date: 2014-12-10
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(Instant,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 in, say, C:\> JAVA.

### Java8Tester.java

```

import java.time.LocalDateTime;
import java.time.ZonedDateTime;

import java.util.Date;

import java.time.Instant;
import java.time.ZoneId;

public class Java8Tester {
    public static void main(String args[]){
        Java8Tester java8tester = new Java8Tester();
        java8tester.testBackwardCompatability();
    }

    public void testBackwardCompatability(){

        //Get the current date

```

```

Date currentDate = new Date();
System.out.println("Current date: " + currentDate);

//Get the instant of current date in terms of milliseconds
Instant now = currentDate.toInstant();
ZoneId currentZone = ZoneId.systemDefault();

LocalDateTime localDateTime = LocalDateTime.ofInstant(now, currentZone);
System.out.println("Local date: " + localDateTime);

ZonedDateTime zonedDateTime = ZonedDateTime.ofInstant(now, currentZone);
System.out.println("Zoned date: " + zonedDateTime);
}
}

```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```

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]

```

## JAVA 8 - 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

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

## Methods

S. No.	Method Name & Description
1	<b>static Base64.Decoder getDecoder()</b> Returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.
2	<b>static Base64.Encoder getEncoder()</b> Returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.
3	<b>static Base64.Decoder getMimeDecoder()</b> Returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.
4	<b>static Base64.Encoder getMimeEncoder()</b> Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.
5	<b>static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)</b> Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators.
6	<b>static Base64.Decoder getUriDecoder()</b> Returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme.
7	<b>static Base64.Encoder getUriEncoder()</b> 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 in say C:/> JAVA.

### Java8Tester.java

```
import java.util.Base64;
import java.util.UUID;
import java.io.UnsupportedEncodingException;

public class HelloWorld {
    public static void main(String args[]){
```

```

try {
    // Encode using basic encoder
    String base64encodedString = Base64.getEncoder().encodeToString("TutorialsPoint?
java8".getBytes("utf-8"));
    System.out.println("Base64 Encoded String (Basic) :" + base64encodedString);

    // Decode
    byte[] base64decodedBytes = Base64.getDecoder().decode(base64encodedString);

    System.out.println("Original String: "+new String(base64decodedBytes, "utf-8"));
    base64encodedString = Base64.getUrlEncoder().encodeToString("TutorialsPoint?
java8".getBytes("utf-8"));
    System.out.println("Base64 Encoded String (URL) :" + base64encodedString);

    StringBuilder stringBuilder = new StringBuilder();

    for (int i = 0; i < 10; ++i) {
        stringBuilder.append(UUID.randomUUID().toString());
    }

    byte[] mimeBytes = stringBuilder.toString().getBytes("utf-8");
    String mimeEncodedString = Base64.getMimeEncoder().encodeToString(mimeBytes);
    System.out.println("Base64 Encoded String (MIME) :"+mimeEncodedString);
} catch (UnsupportedEncodingException e) {
    System.out.println("Error :"+e.getMessage());
}
}
}

```

## Verify the Result

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

```
$javac Java8Tester.java
```

Now run the Java8Tester as follows –

```
$java Java8Tester
```

It should produce the following output –

```

Base64 Encoded String (Basic) :VHV0b3JpYWxzUG9pbmQ/amF2YTg=
Original String: TutorialsPoint?java8
Base64 Encoded String (URL) :VHV0b3JpYWxzUG9pbmQ_amF2YTg=
Base64 Encoded String (MIME) :ZWJjY2YzZWUtYmUwZC00Yjg1LTlkYjUtNWUyMzBlNWl
4ZGQ4ZjE1NGJmMjEtNTdkNi00YzM1LTg4
MzYtNDZlYzNhZDM2NTdkZmQzY2RiNzMtMTU1OC00ZjBmLWVmZGQtM2YyZWU3MDYzZjQwNzVhY
WQ0
ODctZWUyZS00YzM2LWUyZmUtOGVkbmNjMGNmZGM3MTg5YWUyZGQtMzg4MS00M2NkLWI2NDUyZ
jNh
Zjk2OGIxZDU2YzgzODZlYTUtNjlkNC00ZmIyLTkzYTQtMzVlOTFlNjd1Y2E0MDEwNWUyMzBl
mE4
Yy00OTljLTg2NmItMjE3ZTZmMmIyY2NiNzI2MjAwZWQtMjI0NC00YzJhLWJiMGItOTczMDJkM
zIx
NGFkY2QyZmVhODItNmUyOS00MWNjLWFlODItNzdmNzRhYmQ4NGU5ZGQ3ZjY3NzktZjgwYi00M
zlk
LTlmNDgtOTNlNTIwYzIzZDcy

```