**Java 8 Features: A Complete Guide**

Java 8 is going to change the thinking as well as programming style for java programmers. In this article, you will find a complete content to guide you through this new world.

  0

 1

 43

0

[http://mrbool.com/articles/img/handUp.png](http://mrbool.com/java-8-features-a-complete-guide/32550)

[Like (0)](http://mrbool.com/java-8-features-a-complete-guide/32550)

[http://mrbool.com/articles/img/handDown_peq2.png](http://mrbool.com/java-8-features-a-complete-guide/32550)

[(0)](http://mrbool.com/java-8-features-a-complete-guide/32550)

19

**Java 8** is going to change the thinking as well as programming style for java programmers. In Java 8, some features were introduced in the market on 18 march 2014, earlier those were decided to launch in Java 7. Many features are included in Java 8, some of them are Lambda expressions, Java Stream API in util Package, Method References, Default Method, Nashorn, Date/Time API, Base64 etc.

Here is a brief introduction of Java 8 features – A complete guide. But before starting detailed discussion on Java 8, let’s start thinking on Java 8 Benefits.

Why Java 8 is important?

* A lambda expression reduces code complexity.
* Java 8 provides default method in interface so that Lambda expressions can be used easily.
* Adding of Lambda expressions in Java 8, which reduces java code steps hence development through Java 8 takes less time than Java 7 and lower versions.
* Java 8 provides method references which link methods with their names.
* Java 8 adds on streaming features also in which streaming indicates sequence of data. The main need of java.util.stream is to perform pipeline operations for manipulating or searching, mapping, filtering data.

Let’s have a look at Java 8 Activation on Windows, Linux and MAC.

Java 8 activation steps (on Windows, Mac & Linux)

Java Development Kit (JDK) 1.8 (officially named Java SE 8), which is an open source (free for all) from Sun Microsystems. It is now part of Oracle which is the backbone for presenting Java magic.

**What to choose - "JDK" or "JRE"?**

JRE (Java Runtime Environment) is required for running Java magic. JDK (Java Development Kit), which contains JRE & development tools (like compiler and debugger),is required for compiling as well as running Java programs. If you are thinking to [become a Java Programmer](http://mrbool.com/course/java-courses), you should install **JDK**, which contains JRE for compilation and execution of Java code.

***Java 8 activation steps on windows:***

**Step 1**:

Before installing Java 8 on windows please check JDK and JRE. If they have older version then please uninstall JDK and JRE, otherwise multiple versions of JDK and JRE increases confusion .So steps for checking and uninstalling JDK and JRE are: Open "Control Panel” -> Programs -> Program and Features -> Un-install programs starts with "Java", like "Java SE Development Kit" and "Java SE Runtime" etc.

**Step 2**:

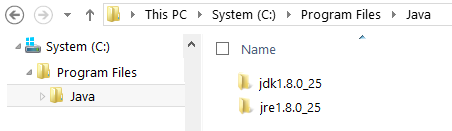
Java SE download site [***http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html***](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)

Please follow the instruction below.

Click on "Java Platform, Standard Edition" then click -> "Java SE 8ux" ->Click the "JDK Download" button for downloading. Now check box for "Accept License Agreement". Select your operating system, e.g., Windows x86 for 32-bit Windows OS or Windows x64 for 64-bit Windows OS.

**Step 3:**

Run the downloaded installer (e.g., "jdk-8uxx-windows-x64.exe"), which installs both the JDK (Java Development Kit) and JRE (Java Runtime). By default, the JDK will be installed in directory "C:\Program Files\Java\jdk1.8.0\_xx", where xx denotes the latest upgrade number; and JRE in



**Image 1:** Showing downloaded JDK

“C:\Program Files\Java\jre1.8.0\_xx".For novices, accept the defaults and simply click "next"..."next"... to install JDK and JRE. This will complete the installation process.

Now you are able to compile and run your java code using of JDK 8 and JRE 8. So if you are not setting environment variable then you have to save and compile your code from ‘**C:\Program Files\Java\jdk1.8.0\_25\bin**’ otherwise you can run your code from all drives.

You can also verify JDK installation by performing the following steps:

1. Go to start menu and open cmd(**Command Prompt**) .
2. Now in cmd go to **C:\Program Files\Java\jdk1.8.0\_25\bin.**
3. In command prompt run the following command.
4. **Javac – version** will verify whether JDK is installed in your machine or not.
5. Output will be **javac** **jdk1.8.0\_25,** if JDK is installed in your machine.

**II) JDK installation on MAC**

***Step 1***: **Check JDK already installed or not:**

In some Mac systems (earlier than **Mac OS X 10.7 Lion**), JDK comes with the OS. For checking whether JDK is installed or not, open a "Terminal" (Go to Utilities then Terminal) and perform these commands:

array0

nbsp;javac-version

If a **javac –version** output looks like jdk1.8.0\_25, then JDK has already been installed. If JDK version is older than 1.7, follow to Step 2 to install the latest JDK;

If command prompt shows like **"command not found",** that means JDK is NOT installed. Follow **"Step 2: Install JDK"**.

***Step 2: Downloading and installing JDK from Oracle site:***

Go to Java SE download site [**http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html**](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)

Under "**Java Platform, Standard Edition**" -> "**Java SE 8ux**" -> Click on "**JDK Download**" button -> Check "**Accept License Agreement**" -> Choose your operating platform, e.g., Mac OS X x64 (**jdk-8uxx-macosx-x64.dmg**).

Then double-click on **jdk-8uxx-macosx-x64.dmg to** install the downloaded Disk Image (DMG) file. Eject the DMG file, if required.

To verify your installation, open a "Terminal" and follow these commands:

Check the version of "javac" (**Compiler**) and "java" (**Java Runtime**)

Checking JDK version by javac –version command:

* **Command: $ javac -version**
* **Output** : **javac 1.x.x\_xx**

Checking JRE version by java –version command:

* **Command: $ java -version**
* **Output:**
* java version "1.x.x\_xxx
* Java(TM) SE Runtime Environment (build 1.x.x\_xx-xxx)

Java HotSpot(TM) Client VM (build 22.1-b02, mixed mode, sharing)

Finding the location of programs "javac"

* **Command: $ which javac**
* **Output : /usr/bin/javac**

Finding the location of programs “java”

* **Command: $ which java**
* **Output : /usr/bin/java**

**III) JDK installation on Ubuntu**

***Step 1***: **Check JDK already installed or not:**

For checking JDK is installed or not, open a "Terminal" and perform the following command:

**array1**

**javac-version**

If a javac –version output look like jdk1.8.0\_25, then JDK has already been installed. If JDK version is older than 1.7, follow to Step 2 to install the latest JDK;

If command prompt shows like **"command not found"** that means JDK is NOT installed. Follow **"Step 2: Install JDK"**.

***Step 2: Downloading and installing JDK from Oracle site:***

Steps for installing JDK

* **$ sudo add-apt-repository ppa:webupd8team/java**
* **$ sudo apt-get update**
* **$ sudo apt-get install oracle-java8-installer**

To verify your installation, open a "Terminal" and follow these commands:

**array2**

**java-version**

java version "1.8.0\_45"

Java(TM) SE Runtime Environment (build 1.8.0\_45-b14)

Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, mixed mode)

***Step 3: Java Code understanding:***

Let us have a look at some sample examples by using Java 8 version

public class JavaTestingExample {

public void display(){

System.out.println("Hello Everyone this is testing program ");

}

public int addInt(int firstNum ,int secNum){

int result ;

result=firstNum+secNum;

return result;

}

public int subInt(int firstNum ,int secNum){

int result ;

result=firstNum-secNum;

return result;

}

public int multiplyInt(int firstNum ,int secNum){

int result ;

result=firstNum\*secNum;

return result;

}

public int divInt(int firstNum ,int secNum){

int result ;

result=firstNum/secNum;

return result;

}

public static void main(String[] args) {

JavaTestingExample javaTestingExample=new JavaTestingExample();

javaTestingExample.display();

System.out.println("additaion of two numbers "+javaTestingExample.addInt(20, 10));

System.out.println("Substraction of two numbers "+javaTestingExample.subInt(20, 10));

System.out.println("Multipication of two numbers "+javaTestingExample.multiplyInt(20, 10));

System.out.println("Division of two numbers "+javaTestingExample.divInt(20, 10));

}

}

***1.1 Java 8: Calculator Example using java 8***

Compile command:

array5

javacJavaTestingExample.java

Running command:

**$java JavaTestingExample**

Output:

**Hello Everyone this is testing program**

**addition of two numbers 30**

**subtraction of two numbers 10**

**Multiplication of two numbers 200**

**Division of two numbers 2**

**Currently Used Companies:** Following table shows companies using Java 8 version.

|  |  |
| --- | --- |
| Java 8 | Linkedin, Google, Twitter |

Java 8 has a lot of new features but it has some disadvantages also. Let us have a look at some of the disadvantages.

**1) Introduction of Parallel Streams reduces performance**

Java 8 has introduced 'parallelism' as one of the most aggressive new feature. The method which implements 'parallelism' is called **parallelStream (),** and it implements collections and streams. The mechanism is to break the job in sub tasks and run in multiple parallel threads. But it reduces the performance significantly.

**2) Overloading problem of Lambda Expressions**

In Java 8 Lambda expressions can be defined as a replacement for anonymous inner classes. A Lambda expression is used to implement functional interface without writing a class or anonymous class. These Lambda expressions are mainly designed to work with a multicore architecture. It supports parallel processing and in turn improves performance. But the main disadvantage is that Lambda expressions can only be used when an interface declares a single method.

**3) Default Methods are creating confusion**

We always know that interfaces only contain method declaration not method body. And there was no mechanism to write method body in an interface. As we know Java does not support multiple inheritance, but it supports multiple inheritance of interfaces, because interfaces do not have any method body. Java 8 introduces a method known as default methods, which enable interfaces to add method body. This is indeed a coolest feature but it also creates confusion to the developers. The concept is new, so it has to be understood clearly before implementation.

**4) Primitives**

Primitives are still present in Java 8. So it is a pain for the developers.

Java 8 features: Description

Now let us discuss about different features introduced in Java 8.

**Lambda Expressions:**

**Introduction:** Lambdas are also known as 'closures' and it is the most awaited feature in Java 8 release. It allows functionality as method argument. Lambdas basically help to replace anonymous inner class.

**Syntax:**

A lambda expression is expressed by the following syntax.

Method parameters -> logic body

Following are some of the important advantages of a lambda expression.

* **Reduce type declaration** - No need to declare type of the method parameter.
* **Reduce parenthesis around parameter** - No need to declare a single parameter in parenthesis.
* **Reduce curly braces** – There is no need to use curly braces in logical part (expression body) if body contains a single statement.
* **Reduce return keyword** -if body has a single expression to return the value at that situation compiler automatically returns the value.

Let us check a simple example where Lambda expression is used to implement Runnable.

**Listing1:** Sample showing Lambda implementation

//Before Java 8:

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("Before Java 8 too much steps for doing this code");

}

}).start();

//Java 8 way:

new Thread( () -> System.out.println("In Java8, Lambda expression changed the way of coding !!") ).start();

**Output:**

Before Java 8 too much steps for doing this code

In Java8, Lambda expression changed the way of coding !!

This example is showing syntax of lambda expression in Java 8. You can write following type of code using lambda expression:

* **(params) -> expression**
* **(params) -> statement**
* **(params) -> { statements }**

for example, if your method does not change/write a parameter and just print something on console, you can write it like this :

**() -> System.out.println("Testing Lambda expresions");**

If your method takes two parameters then you can write them like below :

**(int even, int odd) -> even + odd**

Let us now have a look at how Lambda expression can handle events in Java. The example code shows the implementation of event handler without Lambda expression and with Lambda expression.

**Listing 2 -** Event handling using Java 8 Lambda expressions

// This is the way before Java 8

JButton show = new JButton("Click");

show.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

System.out.println("Testing Event handling without lambda expression ");

}

});

// This is the Java 8 way of implementation

show.addActionListener((e) -> {

System.out.println("Testing event handling using Lambda expression");

})

Now we will check another example where Lambda expressions are used to iterate a list. The example will show the implementation with and without Lambda expression similar to the previous example.

**Listing 3** - Iterating over List using Lambda expressions

//Prior to Java 8 :

List features = Arrays.asList("A", "B", "C", "D");

for (String feature : features) {

System.out.println(feature);

}

//Using Lambda expression in Java 8:

List features = Arrays.asList("A",”B”,”C”,”D”);

features.forEach(n -> System.out.println(n));

If we run the program the output will be as follows.

**Output:**

A

B

C

D

Method Reference

Method references are used to point methods by their name. A method reference is specified by using "::” symbol. A method reference can be used to point these three types of methods.

* Static methods
* Instance methods
* Constructors using new operator

Now let us have a look at a simple example showing method references.

**Listing 4:** Example showing Method References

import java.util.List;

import java.util.ArrayList;

public class ABCClass {

public static void main(String args[]){

List names = new ArrayList();

names.add("Core Java");

names.add("Advance Java");

names.add("Spring Boot");

names.add("Gradle");

names.add("EJB");

names.add("Testing");

names.add("Checking");

names.add("Hibernate");

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

}

}

If we run the program it will show the following output.

**OutPut:**

Core Java

Advance Java

Spring Boot

Gradle

EJB

Testing

Checking

Hibernate

**Functional Interfaces:**

The other core concept of Java 8 is the 'functional interface'. A functional interface can be defined as an interface that contains exactly one abstract method. For example, Runnable is a functional interface because it contains only one abstract method. Following lists shows different functional interfaces.

|  |  |
| --- | --- |
| **Interface** | **Description** |
| BiConsumer | Represents an operation that accepts two input arguments and returns no result. |
| BiFunction | Represents a function that accepts two arguments and produces a result. |
| BinaryOperator | Represents an operation upon two operands of the same type, producing a result of the same type as the operands. |
| BiPredicate | Represents a predicate (boolean-valued function) of two arguments. |
| BooleanSupplier | Represents a supplier of boolean-valued results. |
| Consumer | Represents an operation that accepts a single input argument and returns no result. |
| DoubleBinaryOperator | Represents an operation upon two double-valued operands and producing a double-valued result. |
| DoubleConsumer | Represents an operation that accepts a single double-valued argument and returns no result. |
| DoubleFunction | Represents a function that accepts a double-valued argument and produces a result. |
| DoublePredicate | Represents a predicate (boolean-valued function) of one double-valued argument. |
| DoubleSupplier | Represents a supplier of double-valued results. |
| DoubleToIntFunction | Represents a function that accepts a double-valued argument and produces an int-valued result. |
| DoubleToLongFunction | Represents a function that accepts a double-valued argument and produces a long-valued result. |
| DoubleUnaryOperator | Represents an operation on a single double-valued operand that produces a double-valued result. |
| Function | Represents a function that accepts one argument and produces a result. |
| IntBinaryOperator | Represents an operation upon two int-valued operands and producing an int-valued result. |
| IntConsumer | Represents an operation that accepts a single int-valued argument and returns no result. |
| IntFunction | Represents a function that accepts an int-valued argument and produces a result. |
| IntPredicate | Represents a predicate (boolean-valued function) of one int-valued argument. |
| IntSupplier | Represents a supplier of int-valued results. |
| IntToDoubleFunction | Represents a function that accepts an int-valued argument and produces a double-valued result. |
| IntToLongFunction | Represents a function that accepts an int-valued argument and produces a long-valued result. |
| IntUnaryOperator | Represents an operation on a single int-valued operand that produces an int-valued result. |
| LongBinaryOperator | Represents an operation upon two long-valued operands and producing a long-valued result. |
| LongConsumer | Represents an operation that accepts a single long-valued argument and returns no result. |
| LongFunction | Represents a function that accepts a long-valued argument and produces a result. |
| LongPredicate | Represents a predicate (boolean-valued function) of one long-valued argument. |
| LongSupplier | Represents a supplier of long-valued results. |
| LongToDoubleFunction | Represents a function that accepts a long-valued argument and produces a double-valued result. |
| LongToIntFunction | Represents a function that accepts a long-valued argument and produces an int-valued result. |
| LongUnaryOperator | Represents an operation on a single long-valued operand that produces a long-valued result. |
| ObjDoubleConsumer | Represents an operation that accepts an object-valued and a double-valued argument, and returns no result. |
| ObjIntConsumer | Represents an operation that accepts an object-valued and a int-valued argument, and returns no result. |
| ObjLongConsumer | Represents an operation that accepts an object-valued and a long-valued argument, and returns no result. |
| Predicate | Represents a predicate (boolean-valued function) of one argument. |
| Supplier | Represents a supplier of results. |
| ToDoubleBiFunction | Represents a function that accepts two arguments and produces a double-valued result. |
| ToDoubleFunction | Represents a function that produces a double-valued result. |
| ToIntBiFunction | Represents a function that accepts two arguments and produces an int-valued result. |
| ToIntFunction | Represents a function that produces an int-valued result. |
| ToLongBiFunction | Represents a function that accepts two arguments and produces a long-valued result. |
| ToLongFunction | Represents a function that produces a long-valued result. |
| UnaryOperator | Represents an operation on a single operand that produces a result of the same type as its operand. |

Default Methods: Java 8

Java 8 provides new concept of default method in interface with definition. Default methods functionality is added in Java 8, so that the old functionality will not get affected.

Let us have a look at an example.

**Listing 5:** Example showing Default Method Implementation

public interface ABC {

default void print(){

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

}

}

Multiple Defaults :

public interface ABC {

default void print(){

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

}

}

public interface DEF {

default void print(){

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

}

}

Public class MainClassDemo implements ABC {

Public static void main(String args[]) {

MainClassDemo mainClassDemo=new MainClassDemo();

mainClassDemo. Print();

}

}

If we run the program, the output will be as shown below.

**Output:**

I am a ABC!

Streaming API

Java 8 adds on streaming features also in which streaming indicates sequence of data. Main need of java.util.stream is to perform pipeline operations for manipulating or searching, mapping, filtering data.

Following table shows different types of streams used in Java 8.

**Interfaces and description:**

|  |  |
| --- | --- |
| **Interface** | **Description** |
| BaseStream> | Base interface for streams, which are sequences of elements supporting sequential and parallel aggregate operations. |
| Collector | A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. |
| DoubleStream | A sequence of primitive double-valued elements supporting sequential and parallel aggregate operations. |
| DoubleStream.Builder | A mutable builder for a DoubleStream. |
| IntStream | A sequence of primitive int-valued elements supporting sequential and parallel aggregate operations. |
| IntStream.Builder | A mutable builder for an IntStream. |
| LongStream | A sequence of primitive long-valued elements supporting sequential and parallel aggregate operations. |
| LongStream.Builder | A mutable builder for a LongStream. |
| Stream | A sequence of elements supporting sequential and parallel aggregate operations. |
| Stream.Builder | A mutable builder for a Stream. |

**Classes Summary:**

The following table shows the summary.

|  |  |
| --- | --- |
| **Class** | **Description** |
| Collectors | Implementations of Collector that implement various useful reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc. |
| StreamSupport | Low-level utility methods for creating and manipulating streams. |

Parallel Processing

parallelStream is alternative of stream for parallel processing. Consider below example to print count of empty strings.

List<String> strings = Arrays.asList("a", "b", "c", "d", "e","f", "g",””);

//get count of empty string

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

Arrays.asList("a", "b", "c")

.stream()

.findFirst()

.ifPresent(System.out::println); // a

Optional Classes

Java 8 has introduces a new class known as Optional classes. Optional class can be defined as a container which may or may not contain non-null value. If it contains value then the isPresent () method will return true and the get () method will return the actual value in the container.

**Class declaration:**

Following is the declaration for java.util.Optional class

**Example:** Showing declaration of Optional Classes

public final class Optional<T>

extends Object

**Methods and Description:**

Following are the methods and descriptions.

|  |
| --- |
| * **static Optional empty() :** Returns an empty Optional instance. |
| * **boolean equals(Object obj):** Indicates whether some other object is "equal to" this Optional. |
|  |
| * **Optional filter(Predicate predicate) :** If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional. |
| * **Optional flatMap(Function> mapper):** If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. |
| * **T get():** If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException. |
| * **int hashCode():** Returns the hash code value of the present value, if any, or 0 (zero) if no value is present. |
| * **void ifPresent(Consumer consumer):** If a value is present, invoke the specified consumer with the value, otherwise do nothing. |
| * **boolean isPresent():** Return true if there is a value present, otherwise false. |
| * **Optional map(Function mapper):** If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result. |
| * **static Optional of(T value):** Returns an Optional with the specified present non-null value. |
| * **static Optional ofNullable(T value):** Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional. |
| * **T orElse(T other):** Return the value if present, otherwise return other. |
| * **T orElseGet(Supplier other):** Return the value if present, otherwise invoke other and return the result of that invocation. |
| * **T orElseThrow(Supplier exceptionSupplier):** Return the contained value, if present, otherwise throw an exception to be created by the provided supplier. |
| * **String toString():** Returns a non-empty string representation of this Optional suitable for debugging. |

Java 8 - Nashorn JavaScript Engine

Nashorn is introduced in Java 8 for executing java script code. In Nashorn a lot of new things are add for replacing Rhino java script engine. Nashorm provides 2 to 10 times faster performance as you know it compiles the code in memory and pass the bytecode to JVM.

**Jjs:**

For Nashorn engine, JAVA 8 introduces a new command line tool, jjs to execute java script code at console.

***sample.js***

print('Testing Nashorn Java Script Engine!');

Open console and use the following command.

C:\JAVA>jjs sample.js

See the result.

Testing Nashorn Java Script Engine!

New Date Time API

There were lots of drawbacks in earlier version of Java, so new time has been introduced in Java 8.

**Drawbacks are:**

* Thread safety issue.
* Problem to handle time zone
* Poor way of designing

Following table shows different classes and description.

|  |  |
| --- | --- |
| **Class** | **Description** |
| Clock | A clock providing access to the current instant, date and time using a time-zone. |
| Duration | A time-based amount of time, such as '34.5 seconds'. |
| Instant | An instantaneous point on the time-line. |
| LocalDate | A date without a time-zone in the ISO-8601 calendar system, such as 2007-12-03. |
| LocalDateTime | A date-time without a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30. |
| LocalTime | A time without time-zone in the ISO-8601 calendar system, such as 10:15:30. |
| MonthDay | A month-day in the ISO-8601 calendar system, such as --12-03. |
| OffsetDateTime | A date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00. |
| OffsetTime | A time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 10:15:30+01:00. |
| Period | A date-based amount of time in the ISO-8601 calendar system, such as '2 years, 3 months and 4 days'. |
| Year | A year in the ISO-8601 calendar system, such as 2007. |
| YearMonth | A year-month in the ISO-8601 calendar system, such as 2007-12. |
| ZonedDateTime | A date-time with a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00 Europe/Paris. |
| ZoneId | A time-zone ID, such as Europe/Paris. |
| ZoneOffset | A time-zone offset from Greenwich/UTC, such as +02:00. |

**Enum Summary:**

* **DayOfWeek:** A day-of-week, such as 'Tuesday'.
* **Month**: A month-of-year, such as 'July'.

**Exception Summary:**

**DateTimeException: Exception** used to indicate a problem while calculating a date-time

Base64

Now in Java 8 Base64 has inbuilt encoder and decoder. In java 8 we can use three type of Base64 encoding.

1. Simple
2. URL
3. MIME

Let us have a look at different methods and descriptions.

|  |
| --- |
| * **static Base64.Decoder getDecoder():** Returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme. |
| * **static Base64.Encoder getEncoder():** Returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme. |
| * **static Base64.Decoder getMimeDecoder():** Returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme. |
| * **static Base64.Encoder getMimeEncoder():** Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme. |
| * **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. |
| * **static Base64.Decoder getUrlDecoder():** Returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme. |
| * **static Base64.Encoder getUrlEncoder():** Returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme. |

Now let us check an example below.

**Example** : Base64 using in Java Code

import java.util.Base64;

import java.util.UUID;

public class HelloWorld{

public static void main(String []args) throws Exception{

String base64encodedString = Base64.getEncoder().encodeToString("Testing Base64".getBytes("utf-8"));

System.out.println("Base64 Encoded :" + base64encodedString);

// Decode

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

System.out.println("Original String: "+new String(base64decodedBytes, "utf-8"));

}

}

**OUTPUT:**

array26

javacHelloWorld.java

array26

java-Xmx128M-Xms16MHelloWorld

Base64Encoded:VGVzdGluZyBCYXNlNjQ=

OriginalString:TestingBase64

Conclusion

Read more: <http://mrbool.com/java-8-features-a-complete-guide/32550#ixzz3iPjYk0od>