**What new features were added in Java 8?**

Java 8 ships with several new features but the most significant are the following:

* **Lambda Expressions**− a new language feature allowing treating actions as objects
* **Method References** − enable defining Lambda Expressions by referring to methods directly using their names
* ***Optional*** − special wrapper class used for expressing optionality
* **Functional Interface** – an interface with maximum one abstract method, implementation can be provided using a Lambda Expression
* **Default methods** − give us the ability to add full implementations in interfaces besides abstract methods
* **Nashorn, JavaScript Engine** − Java-based engine for executing and evaluating JavaScript code
* ***Stream* API** − a special iterator class that allows processing collections of objects in a functional manner
* **Date API** − an improved, immutable JodaTime-inspired Date API

Along with these new features, lots of feature enhancements are done under-the-hood, at both compiler and JVM level.

**3. Method References**

**What is a method reference?**

A method reference is a Java 8 construct that can be used for referencing a method without invoking it. It is used for treating methods as Lambda Expressions. They only work as syntactic sugar to reduce the verbosity of some lambdas. This way, the following code:

(o) -> o.toString();

can become:

Object::toString();

A method reference can be identified by a double colon separating a class or object name and the name of the method. It has different variations such as

constructor reference:

String::new;

Static method reference:

String::valueOf;

Bound instance method reference:

str::toString;

Unbound instance method reference:

String::toString;

You can read a detailed description of method references with full examples by following this link and this one.

**What is the meaning of String::valueOf expression?**

It is a static method reference to the *valueOf* method of the *String* class.

**4. Optional**

**What is *Optional*? How can it be used?**

*Optional* is a new class in Java 8 that encapsulates an optional value i.e. a value that is either there or not. It is a wrapper around an object, and you can think of it as a container of zero or one element.

*Optional* has a special *Optional.empty()* value instead of wrapped *null*. Thus it can be used instead of a nullable value to get rid of *NullPointerException* in many cases.

The main purpose of *Optional*, as designed by its creators, was to be a return type of methods that previously would return *null*. Such methods would require you to write boilerplate code to check the return value and sometimes could forget to do a defensive check. In Java 8, an *Optional* return type explicitly requires you to handle null or non-null wrapped values differently.

For instance, the *Stream.min()* method calculates the minimum value in a stream of values. But what if the stream is empty? If it was not for *Optional*, the method would return *null*or throw an exception.

But it returns an *Optional* value which may be *Optional.empty()* (the second case). This allows us to easily handle such case:

It’s worth noting that *Optional* is not a general purpose class like *Option* in Scala. It is not recommended to be used as a field value in entity classes, which is clearly indicated by it not implementing the *Serializable* interface.

**5. Functional Interfaces**

**Describe some of the functional interfaces in the standard library.**

There are a lot of functional interfaces in the *java.util.function* package, the more common ones include but not limited to:

* *Function* – it takes one argument and returns a result
* *Consumer* – it takes one argument and returns no result (represents a side effect)
* *Supplier* – it takes not argument and returns a result
* *Predicate* – it takes one argument and returns a boolean
* *BiFunction* – it takes two arguments and returns a result
* *BinaryOperator* – it is similar to a *BiFunction*, taking two arguments and returning a result. The two arguments and the result are all of the same types
* *UnaryOperator* – it is similar to a *Function*, taking a single argument and returning a result of the same type

For more on functional interfaces, see the article [“Functional Interfaces in Java 8”](https://www.baeldung.com/java-8-functional-interfaces).

**What is a functional interface? What are the rules of defining a functional interface?**

A functional interface is an interface with no more, no less but one single abstract method (*default* methods do not count).

Where an instance of such interface is required, a Lambda Expression can be used instead. More formally put: *Functional interfaces* provide target types for lambda expressions and method references.

The arguments and return type of such expression directly match those of the single abstract method.

For instance, the *Runnable* interface is a functional interface, so instead of:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Thread thread = new Thread(new Runnable() {      public void run() {          System.out.println("Hello World!");      }  }); |

you could simply do:

|  |  |
| --- | --- |
| 1 | Thread thread = new Thread(() -> System.out.println("Hello World!")); |

Functional interfaces are usually annotated with the *@FunctionalInterface* annotation – which is informative and does not affect the semantics.

**6. Default Method**

**Q1. What is a default method and when do we use it?**

A default method is a method with an implementation – which can be found in an interface.

We can use a default method to add a new functionality to an interface while maintaining backward compatibility with classes that are already implementing the interface:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface Vehicle {      public void move();      default void hoot() {          System.out.println("peep!");      }  } |

Usually, when a new abstract method is added to an interface, all implementing classes will break until they implement the new abstract method. In Java 8, this problem has been solved by the use of default method.

For example, *Collection* interface does not have *forEach* method declaration. Thus, adding such method would simply break the whole collections API.

Java 8 introduces default method so that *Collection* interface can have a default implementation of *forEach*method without requiring the classes implementing this interface to implement the same.

**Q2. Will the following code compile?**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @FunctionalInterface  public interface Function2<T, U, V> {      public V apply(T t, U u);        default void count() {          // increment counter      }  } |

Yes. The code will compile because it follows the functional interface specification of defining only a single abstract method. The second method, *count*, is a default method that does not increase the abstract method count.

**7. Lambda Expressions**

**Q1. What is a Lambda Expression and what is it used for**

In very simple terms, a lambda expression is a function that can be referenced and passed around as an object.

Lambda expressions introduce functional style processing in Java and facilitate the writing of compact and easy-to-read code.

Because of this, lambda expressions are a natural replacement for anonymous classes as method arguments. One of their main uses is to define inline implementations of functional interfaces.

**Q2. Explain the syntax and characteristics of a Lambda Expression**

A lambda expression consists of two parts: the parameter part and the expressions part separated by a forward arrow as below:

|  |  |
| --- | --- |
| 1 | params -> expressions |

Any lambda expression has the following characteristics:

* **Optional type declaration** – when declaring the parameters on the left-hand side of the lambda, we don’t need to declare their types as the compiler can infer them from their values. So *int param -> …* and *param ->…*are all valid
* **Optional parentheses** – when only a single parameter is declared, we don’t need to place it in parentheses. This means *param -> …* and *(param) -> …* are all valid. But when more than one parameter is declared, parentheses are required
* **Optional curly braces** – when the expressions part only has a single statement, there is no need for curly braces. This means that *param – > statement* and *param – > {statement;}* are all valid. But curly braces are required when there is more than one statement
* **Optional return statement** – when the expression returns a value and it is wrapped inside curly braces, then we don’t need a return statement. That means *(a, b) – > {return a+b;}* and *(a, b) – > {a+b;}* are both valid

To read more about Lambda expressions, follow [this link](https://www.tutorialspoint.com/java8/java8_lambda_expressions.htm) and [this one](https://www.baeldung.com/java-8-lambda-expressions-tips).

**8. Nashorn Javascript**

**Q1. What is Nashorn in Java8?**

[Nashorn](https://www.baeldung.com/java-nashorn) is the new Javascript processing engine for the Java platform that shipped with Java 8. Until JDK 7, the Java platform used Mozilla Rhino for the same purpose. as a Javascript processing engine.

Nashorn provides better compliance with the ECMA normalized JavaScript specification and better runtime performance than its predecessor.

**Q2. What is jjs?**

In Java 8, *jjs* is the new executable or command line tool used to execute Javascript code at the console.

**9. Streams**

**Q1. What is a stream? How does it differ from a collection?**

In simple terms, a stream is an iterator whose role is to accept a set of actions to apply on each of the elements it contains.

*The stream* represents a sequence of objects from a source such as a collection, which supports aggregate operations. They were designed to make collection processing simple and concise. Contrary to the collections, the logic of iteration is implemented inside the stream, so we can use methods like *map* and *flatMap* for performing a declarative processing.

Another difference is that the *Stream* API is fluent and allows pipelining:

|  |  |
| --- | --- |
| 1  2  3  4 | int sum = Arrays.stream(new int[]{1, 2, 3})    .filter(i -> i >= 2)    .map(i -> i \* 3)    .sum(); |

And yet another important distinction from collections is that streams are inherently lazily loaded and processed.

**Q2. What is the difference between intermediate and terminal operations?**

Stream operations are combined into pipelines to process streams. All operations are either intermediate or terminal.

Intermediate operations are those operations that return *Stream* itself allowing for further operations on a stream.

These operations are always lazy, i.e. they do not process the stream at the call site, an intermediate operation can only process data when there is a terminal operation. Some of the intermediate operations are *filter*, *map* and *flatMap*.

Terminal operations terminate the pipeline and initiate stream processing. The stream is passed through all intermediate operations during terminal operation call. Terminal operations include *forEach*, *reduce, Collect* and *sum*.

To drive this point home, let us look at an example with side effects:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static void main(String[] args) {      System.out.println("Stream without terminal operation");        Arrays.stream(new int[] { 1, 2, 3 }).map(i -> {          System.out.println("doubling " + i);          return i \* 2;      });        System.out.println("Stream with terminal operation");          Arrays.stream(new int[] { 1, 2, 3 }).map(i -> {              System.out.println("doubling " + i);              return i \* 2;      }).sum();  } |

The output will be as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Stream without terminal operation  Stream with terminal operation  doubling 1  doubling 2  doubling 3 |

As you can see, the intermediate operations are only triggered when a terminal operation exists.

**Q3. What is the difference between *map* and *flatMap* stream operation?**

There is a difference in signature between *map* and *flatMap*. Generally speaking, a *map* operation wraps its return value inside its ordinal type while *flatMap* does not.

For example, in *Optional*, a *map* operation would return *Optional<String>* type while *flatMap* would return *String*type.

So after mapping, one needs to unwrap (read “flatten”) the object to retrieve the value whereas, after flat mapping, there is no such need as the object is already flattened. The same concept is applied to mapping and flat mapping in *Stream*.

Both *map* and *flatMap* are intermediate stream operations that receive a function and apply this function to all elements of a stream.

The difference is that for the *map*, this function returns a value, but for *flatMap*, this function returns a stream. The *flatMap* operation “flattens” the streams into one.

Here’s an example where we take a map of users’ names and lists of phones and “flatten” it down to a list of phones of all the users using *flatMap*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Map<String, List<String>> people = new HashMap<>();  people.put("John", Arrays.asList("555-1123", "555-3389"));  people.put("Mary", Arrays.asList("555-2243", "555-5264"));  people.put("Steve", Arrays.asList("555-6654", "555-3242"));    List<String> phones = people.values().stream()    .flatMap(Collection::stream)      .collect(Collectors.toList()); |

**Q4. What is stream pipelining in Java 8?**

Stream pipelining is the concept of chaining operations together. This is done by splitting the operations that can happen on a stream into two categories: intermediate operations and terminal operations.

Each intermediate operation returns an instance of Stream itself when it runs, an arbitrary number of intermediate operations can, therefore, be set up to process data forming a processing pipeline.

There must then be a terminal operation which returns a final value and terminates the pipeline.

**10. Java 8 Date and Time API**

**Q1. Tell us about the new Date and Time API in Java 8**

A long-standing problem for Java developers has been the inadequate support for the date and time manipulations required by ordinary developers.

The existing classes such as *java.util.Date* and *SimpleDateFormatter* aren’t thread-safe, leading to potential concurrency issues for users.

Poor API design is also a reality in the old Java Data API. Here’s just a quick example – years in *java.util.Date* start at 1900, months start at 1, and days start at 0 which is not very intuitive.

These issues and several others have led to the popularity of third-party date and time libraries, such as Joda-Time.

In order to address these problems and provide better support in JDK, a new date and time API, which is free of these problems, has been designed for Java SE 8 under the package *java.time*.

**Why do we need change to Java again?**

Oracle Corporation has introduced a lot of new concepts in Java SE 8 to introduce the following benefits:

* **To Utilize Current Multi-Core CPUs Efficiently**
* Recently, we can observe drastic changes in Hardware. Now-a-days, all systems are using Multi-Core CPUs (2,4,8,16-Core etc.) to deploy and run their Applications. We need new Programming Constructs in Java to utilize these Multi-Core Processors efficiently to develop Highly Concurrently and Highly Scalable applications.
* **To Utilize FP Features**
* Oracle Corporation has introduced a lot of FP(Functional Programming) concepts as part of Java SE 8 to utilize the advantages of FP.

**Java SE 8 New Features?**

* Lambda Expressions
* Functional Interfaces
* [Stream API](https://www.journaldev.com/2774/java-8-stream)
* Date and Time API
* [Interface Default Methods and Static Methods](https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method)
* Spliterator
* Method and Constructor References
* Collections API Enhancements
* Concurrency Utils Enhancements
* Fork/Join Framework Enhancements
* Internal Iteration
* Parallel Array and Parallel Collection Operations
* Optional
* Type Annotations and Repeatable Annotations
* Method Parameter Reflection
* Base64 Encoding and Decoding
* IO and NIO2 Enhancements
* Nashorn JavaScript Engine
* javac Enhancements
* JVM Changes
* Java 8 Compact Profiles: compact1,compact2,compact3
* JDBC 4.2
* JAXP 1.6
* Java DB 10.10
* Networking
* Security Changes

**Advantages of Java SE 8 New Features?**

We can get the following benefits from Java SE 8 New Features:

* More Concise and Readable code
* More Reusable code
* More Testable and Maintainable Code
* Highly Concurrent and Highly Scalable Code
* Write Parallel Code
* Write Database Like Operations
* Better Performance Applications
* More Productive code

**What is Lambda Expression?**

Lambda Expression is an anonymous function, which accepts a set of input parameters and returns results.

Lambda Expression is a block of code without any name, with or without parameters and with or without results. This block of code is executed on demand.

**What are the three parts of a Lambda Expression? What is the type of Lambda Expression?**

A Lambda Expression contains 3 parts:

* + - * Parameter List

A Lambda Expression can contain zero or one or more parameters. It is optional.

* Lambda Arrow Operator
  + - * 1. “->” is known as Lambda Arrow operator. It separates parameters list and body.
* Lambda Expression Body

The type of “Journal Dev” is java.lang.String. The type of “true” is Boolean. In the same way, what is the type of a Lambda Expression?

The Type of a Lambda Expression is a [Functional Interface](https://www.journaldev.com/2763/java-8-functional-interfaces).

Example:- What is the type of the following Lambda Expression?

()->System.out.println("Hello World");

This Lambda Expression does not have parameters and does return any results. So it’s type is “java.lang.Runnable” Functional Interface.

**What is a Functional Interface? What is SAM Interface?**

A Functional Interface is an interface, which contains one and only one abstract method. Functional Interface is also know as SAM Interface because it contains only one abstract method.

SAM Interface stands for Single Abstract Method Interface. Java SE 8 API has defined many Functional Interfaces.

**Is this possible to define our own Functional Interface? What is @FunctionalInterface? What are the rules to define a Functional Interface?**

Yes, it is possible to define our own Functional Interfaces. We use Java SE 8’s @FunctionalInterface annotation to mark an interface as Functional Interface.

We need to follow these rules to define a Functional Interface:

* Define an interface with one and only one abstract method.
* We cannot define more than one abstract method.
* Use @FunctionalInterface annotation in interface definition.
* We can define any number of other methods like Default methods, Static methods.
* If we override java.lang.Object class’s method as an abstract method, which does not count as an abstract method.

**Is @FunctionalInterface annotation mandatory to define a Functional Interface? What is the use of @FunctionalInterface annotation? Why do we need Functional Interfaces in Java?**

It is not mandatory to define a Functional Interface with @FunctionalInterface annotation. If we don’t want, We can omit this annotation. However, if we use it in Functional Interface definition, Java Compiler forces to use one and only one abstract method inside that interface.

Why do we need Functional Interfaces? The type of a Java SE 8’s Lambda Expression is a Functional Interface. Whereever we use Lambda Expressions that means we are using Functional Interfaces.

**When do we go for Java 8 Stream API? Why do we need to use Java 8 Stream API in our projects?**

When our Java project wants to perform the following operations, it’s better to use [Java 8 Stream](https://www.journaldev.com/2774/java-8-stream) API to get lot of benefits:

* When we want perform Database like Operations. For instance, we want perform groupby operation, orderby operation etc.
* When want to Perform operations Lazily.
* When we want to write Functional Style programming.
* When we want to perform Parallel Operations.
* When want to use Internal Iteration
* When we want to perform Pipelining operations.
* When we want to achieve better performance.

**Explain Differences between Collection API and Stream API?**

|  |  |  |
| --- | --- | --- |
| S.NO. | COLLECTION API | STREAM API |
| 1. | It’s available since Java 1.2 | It is introduced in Java SE8 |
| 2. | It is used to store Data(A set of Objects). | It is used to compute data(Computation on a set of Objects). |
| 3. | We can use both Spliterator and Iterator to iterate elements. We can use [forEach](https://www.journaldev.com/13941/java-foreach-java-8-foreach) to performs an action for each element of this stream. | We can’t use Spliterator or Iterator to iterate elements. |
| 4. | It is used to store limited number of Elements. | It is used to store either Limited or Infinite Number of Elements. |
| 5. | Typically, it uses Internal Iteration concept to iterate Elements. | It uses External Iteration to iterate Elements. |
| 6. | Collection Object is constructed Eagerly. | Stream Object is constructed Lazily. |
| 7. | We add elements to Collection object only after it is computed completely. | We can add elements to Stream Object without any prior computation. That means Stream objects are computed on-demand. |
| 8. | We can iterate and consume elements from a Collection Object at any number of times. | We can iterate and consume elements from a Stream Object only once. |

**What is Spliterator in Java SE 8?Differences between Iterator and Spliterator in Java SE 8?**

Spliterator stands for Splitable Iterator. It is newly introduced by Oracle Corporation as part Java SE 8.

Like Iterator and ListIterator, It is also one of the Iterator interface.

|  |  |  |
| --- | --- | --- |
| S.NO. | SPLITERATOR | ITERATOR |
| 1. | It is introduced in Java SE 8. | It is available since Java 1.2. |
| 2. | Splitable Iterator | Non-Splitable Iterator |
| 3. | It is used in Stream API. | It is used for Collection API. |
| 4. | It uses Internal Iteration concept to iterate Streams. | It uses External Iteration concept to iterate Collections. |
| 5. | We can use Spliterator to iterate Streams in Parallel and Sequential order. | We can use Iterator to iterate Collections only in Sequential order. |
| 6. | We can get Spliterator by calling spliterator() method on Stream Object. | We can get Iterator by calling iterator() method on Collection Object. |
| 7. | Important Method: tryAdvance() | Important Methods: next(), hasNext() |

**What is Optional in Java 8? What is the use of Optional? Advantages of Java 8 Optional?**

**Optional:**

Optional is a final Class introduced as part of Java SE 8. It is defined in java.util package.

It is used to represent optional values that is either exist or not exist. It can contain either one value or zero value. If it contains a value, we can get it. Otherwise, we get nothing.

It is a bounded collection that is it contains at most one element only. It is an alternative to “null” value.

**Main Advantage of Optional is:**

* It is used to avoid null checks.
* It is used to avoid “NullPointerException”.

**What is Type Inference? Is Type Inference available in older versions like Java 7 and Before 7 or it is available only in Java SE 8?**

Type Inference means determining the Type by compiler at compile-time.

It is not new feature in Java SE 8. It is available in Java 7 and before Java 7 too.

**Before Java 7:-**

Let us explore Java arrays. Define a String of Array with values as shown below:

String str[]={"Java 7","Java 8","Java 9"};

Here we have assigned some String values at right side, but not defined it’s type. Java Compiler automatically infers it’s type and creates a String of Array.

**Java 7:-**

Oracle Corporation has introduced “Diamond Operator” new feature in Java SE 7 to avoid unnecessary Type definition in Generics.

Map<String,List<Customer>> customerInfoByCity =newHashMap<>();

Here we have not defined Type information at right side, simply defined Java SE 7’s Diamond Operator “”.

**Java SE 8:-**

Oracle Corporation has enhanced this Type Inference concept a lot in Java SE 8. We use this concept to define Lambda Expressions, Functions, Method References etc.

ToIntBiFunction<Integer,Integer> add =(a,b)-> a + b;

Here Java Compiler observes the type definition available at left-side and determines the type of Lambda Expression parameters a and b as Integers.

What are the new features introduced in JAVA 8?

There are dozens of features added to Java 8, the most significant ones are mentioned below −

* **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.

Along with these new featuers, lots of feature enhancements are done under-the-hood, at both compiler and JVM level.

How will you sort a list of string using Java 8 lambda expression?

Following code sorts a list of string using Java 8 lambda expression:

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

What are the characteristics of a Java 8 lambda expression?

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.

**Why lambda expression is to be used?**

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.

**What kind of variable you can access in an lambda expression??**

Using lambda expression, you can refer to 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.

**What are method references?**

Method references help to point to methods by their names. A method reference is described using :: (double colon) symbol. A method reference can be used to point the following types of methods −

* Static methods
* Instance methods
* Constructors using new operator (TreeSet::new)

**Explain the System.out::println expression.**

System.out::println method is a static method reference to println method of out object of System class.

**What are 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.

**What are default methods?**

With java 8, an interface can have default implementation of a function in interfaces.

**What are static default methods?**

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

publicinterface vehicle {defaultvoidprint(){System.out.println("I am a vehicle!");}staticvoid blowHorn(){System.out.println("Blowing horn!!!");}}

**How will you call a default method of an interface in a class?**

Using super keyword along with interface name.

interfaceVehicle{defaultvoidprint(){System.out.println("I am a vehicle!");}}classCarimplementsVehicle{publicvoidprint(){Vehicle.super.print();}}

**How will you call a static method of an interface in a class?**

Using name of the interface.

interfaceVehicle{staticvoid blowHorn(){System.out.println("Blowing horn!!!");}}classCarimplementsVehicle{publicvoidprint(){Vehicle.blowHorn();}}

**What is streams in Java 8?**

Stream represents a sequence of objects from a source, which supports aggregate operations.

**What is stream pipelining in Java 8?**

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.

**What is the difference between Collections and Stream in Java8 ?**

Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

**What is the purpose of forEach method of stream in java 8?**

Stream has provided a new method 'forEach' to iterate each element of the stream.

**How will you print 10 random numbers using forEach of java 8?**

The following code segment shows how to print 10 random numbers using forEach.

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

**What is the purpose of map method of stream in java 8?**

The 'map' method is used to map each element to its corresponding result.

**How will you print unique squares of numbers in java 8?**

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

**What is the purpose of filter method of stream in java 8?**

The 'filter' method is used to eliminate elements based on a criteria.

**How will you print count of empty strings in java 8?**

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

**What is the purpose of limit method of stream in java 8?**

The 'limit' method is used to reduce the size of the stream.

**How will you print 10 random numbers in java 8?**

The following code segment shows how to print 10 random numbers.

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

**What is the purpose of sorted method of stream in java 8?**

The 'sorted' method is used to sort the stream.

**How will you print 10 random numbers in a sorted order in java 8?**

The following code segment shows how to print 10 random numbers in a sorted order.

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

**What is Parallel Processing in Java 8?**

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 stringint count = strings.parallelStream().filter(string−>string.isEmpty()).count();//It is very easy to switch between sequential and parallel streams.

**What are collectors in Java 8?**

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);

**What are Statistics collectors in Java 8?**

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

**How will you get the highest number present in a list using Java 8?**

Following code will print the highest number present in a list.

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());

**How will you get the lowest number present in a list using Java 8?**

Following code will print the highest number present in a list.

List<Integer> numbers =Arrays.asList(3,2,2,3,7,3,5);IntSummaryStatistics stats = integers.stream().mapToInt((x)−> x).summaryStatistics();System.out.println("Lowest number in List : "+ stats.getMin());

**How will you get the sum of all numbers present in a list using Java 8?**

Following code will print the sum of all numbers present in a list.

List<Integer> numbers =Arrays.asList(3,2,2,3,7,3,5);IntSummaryStatistics stats = integers.stream().mapToInt((x)−> x).summaryStatistics();System.out.println("Sum of all numbers : "+ stats.getSum());

**How will you get the average of all numbers present in a list using Java 8?**

Following code will print the average of all numbers present in a list.

List<Integer> numbers =Arrays.asList(3,2,2,3,7,3,5);IntSummaryStatistics stats = integers.stream().mapToInt((x)−> x).summaryStatistics();System.out.println("Average of all numbers : "+ stats.getAverage());

**What is Optional in Java8?**

Optional is a container object which is 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.

**What is Nashorn in Java8?**

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 invokedynamics feature, introduced in Java 7 to improve performance.

**What is jjs in JAVA8?**

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

**Can you execute javascript code from java 8 code base?**

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

**What is local datetime API in JAVA8?**

Local − Simplified date-time API with no complexity of timezone handling.

**What is zoned datetime API in JAVA8?**

Zoned − Specialized date-time API to deal with various timezones.

**What is chromounits in java8?**

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

**How will you get the current date using local datetime api of java8?**

Following code gets the current date using local datetime api −

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

**How will you add 1 week to current date using local datetime api of java8?**

Following code adds 1 week to current date using local datetime api −

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

**How will you add 1 month to current date using local datetime api of java8?**

Following code adds 1 month to current date using local datetime api:

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

**How will you add 1 year to current date using local datetime api of java8?**

Following code adds 1 year to current date using local datetime api −

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

**How will you add 10 years to current date using local datetime api of java8?**

Following code adds 10 years to current date using local datetime api −

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

**How will you get next tuesday using java8?**

Following code gets next tuesday using java8 −

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

**How will you get second saturday of next month using java8?**

Following code gets second saturday of next month using java8 −

//get the second saturday of next monthLocalDate 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);

**How will you get the instant of current date in terms of milliseconds using java8?**

Following code gets the instant of current date in terms of milliseconds −

//Get the instant of current date in terms of millisecondsInstant now = currentDate.toInstant();

**How will you get the instant of local date time using time in of milliseconds using java8?**

Following code gets the instant of local date time using time in of milliseconds −

Instant now = currentDate.toInstant();ZoneId currentZone =ZoneId.systemDefault();LocalDateTime localDateTime =LocalDateTime.ofInstant(now, currentZone);System.out.println("Local date: "+ localDateTime);

**How will you get the instant of zoned date time using time in of milliseconds using java8?**

Following code gets the instant of zoned date time using time in of milliseconds −

Instant now = currentDate.toInstant();ZoneId currentZone =ZoneId.systemDefault();ZonedDateTime zonedDateTime =ZonedDateTime.ofInstant(now, currentZone);System.out.println("Zoned date: "+ zonedDateTime);

**Which class implements a decoder for decoding byte data using the Base64 encoding scheme in Java8?**

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.

**Which class implements an encoder for encoding byte data using the Base64 encoding scheme in Java8?**

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.

# **What is the difference in hashmap implementation in Java 8 and pre-Java 8?**

Now, prior to J8 this bucket was just a linked list. I.e. if you did have some clashes in your data, then looking between them was a normal linear search on that bucket. What J8 changed was that this became adaptive - e.g. if there’s only a handful of clashing items in a bucket it’s still the same. But once there are more than a predetermined amount of clashing items in the same bucket - it is turned into a balance binary tree instead.

The benefit of this is that if the hashing formula isn’t great and tends to cause many clashes, your average seek time is still not as bad as a O(N) linear search. It’s closer to the O(log N) of a binary search instead.