# **Java 8 features:**

==============================================================

Link:https://hackr.io/blog/java-8-interview-questions

The most distinct features of Java 8 include:

#### Functional Interfaces

#### Lambda Expressions

#### Optional Class

#### Inference

#### Spliterator

#### New Stream API

#### New Data/Time API

#### Method References

# Lambda Expressions

## What is a Lambda Expression?

* A Lambda Expression is an anonymous function that accepts different sets of input parameters and offers a variety of outputs. In essence, it is an anonymous method. It is useful because it offers a functional interface.

## Why is Lambda Expression considered a block of code?

* Lambda Expression is considered a block of code because it has no name and can exist with or without parameters and results.

## What are the different parts of Lambda Expression?

* Parameter List
* Lambda Body Expression
* Lambda Arrow Operator

## What are the features of a lambda expression?

* It has no specific parameters.
* It returns any kind of result.

## **Program1:**

package test;

public class LambdaEx1 {

public static void main(String[] args) {

// multiple parameters

If1 if1 = (x, y) -> {

return x + y;

};

int cal = if1.cal(1, 2);

System.*out*.println(cal);

// single parameter

If2 if2 = () -> System.*out*.println("Hai");

if2.hello();

}

}

@FunctionalInterface

interface If1 {

public int cal(int x, int y);

}

@FunctionalInterface

interface If2 {

public void hello();

}

**Out Put:**

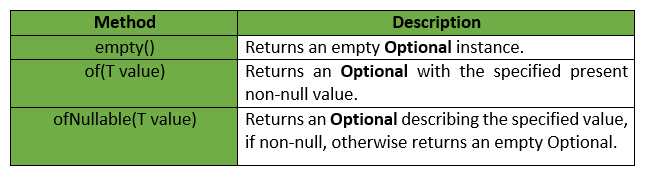
3

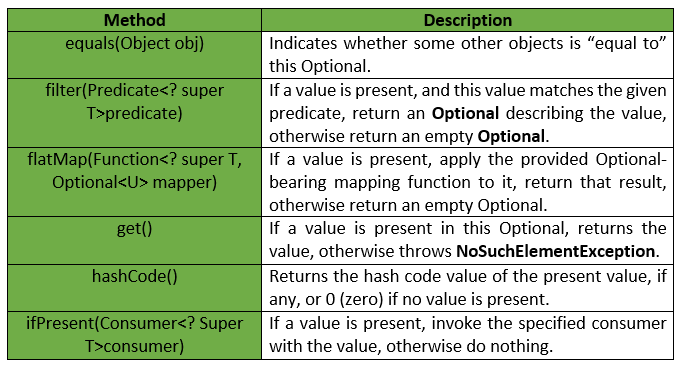
Hai

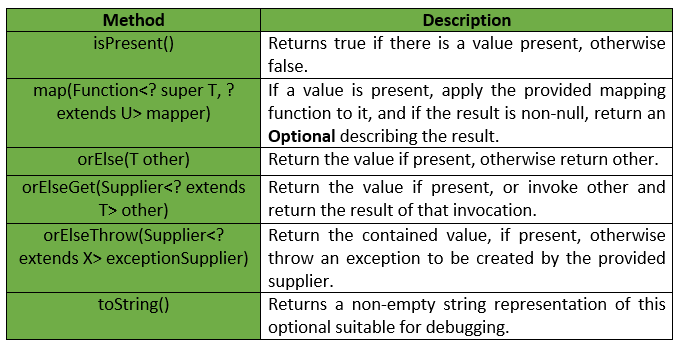
# **Optional Class**

## **1)Define Optional in Java SE 8?**

* Optional is a **final class** that is introduced as an integral part of Java SE8.
* It is a java.util package that can have either one value or zero value.
* A container object which may or may not contain a non-null value.
* If a value is present, isPresent() will return true and get() will return the value.







## 2) What is the use of Optional in Java SE 8?

* Optional in Java SE 8 is used for representing optional values that exist or does not exist.
* It does not support too many null checks and NullPointerException.
* In addition to that, it avoids the runtime NullPointerExceptions and helps in developing cleaner Java APIs.

## 3) What are the advantages of Optional?

* Optional has the following advantages.
* It helps in avoiding the null checks.
* It is used for avoiding “**NullPointerException**”.

## 4) What is Type Inference in Java 8?

* Type Interface refers to the compiler’s ability to determine the type of parameters when the method is called, i.e., at compile time.

# **Functional Interfaces**

## **1) What is the Default Method?**

* A default method adds functionality to an interface without breaking down the implementing class, and offers backwards compatibility.
* Before Java 8, interfaces could have only abstract methods.
* The implementation of these methods has to be provided in a separate class.
* So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface.
* To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

## **Program 1:**

// A simple program to Test Interface default

// methods in java

interface TestInterface

{

// abstract method

public void square(int a);

// default method

default void show(){

System.out.println("Default Method Executed");

}

}

class TestClass implements TestInterface

{

// implementation of square abstract method

public void square(int a)

{

System.out.println(a\*a);

}

public static void main(String args[])

{

TestClass d = new TestClass();

d.square(4);

// default method executed

d.show();

}

}

## **Program 2:**

interface TestInterface

{

// abstract method

public void square (int a);

// static method

static void show()

{

System.out.println("Static Method Executed");

}

}

class TestClass implements TestInterface

{

// Implementation of square abstract method

public void square (int a)

{

System.out.println(a\*a);

}

public static void main(String args[])

{

TestClass d = new TestClass();

d.square(4);

// Static method executed

TestInterface.show();

}

}

# **Stream API**

## **1) Define Stream Pipelining.**

* Stream API is used to process collections of objects.
* A stream is a **sequence of objects** that supports various methods which can be pipelined to produce **the desired result**.
* Stream Pipelining in Java SE 8 is used for **chaining** **operations** **together** by **splitting** the operations that can happen on one stream.

## **2)what are the features of Java stream api?**

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.
* The two categories of Stream Pipelining are
* **1)intermediate operations 2)terminal operations.**

## **3)Intermediate like**

1. Limit(long n)
2. skip (long n)
3. Distinct ()
4. Filter (Predicate)
5. Map (Function)

## 4)Terminal Operations like

1. Max
2. Min
3. AnyMatch
4. AllMatch
5. Reduce
6. ToArray
7. Count

**Intermediate Operations:**

* **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
  List number = Arrays.asList(2,3,4,5);  
  List square = number.stream().map(x->x\*x).collect(Collectors.toList());
* **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
  List names = Arrays.asList("Reflection","Collection","Stream");  
  List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
* **sorted:** The sorted method is used to sort the stream.  
  List names = Arrays.asList("Reflection","Collection","Stream");  
  List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

**2) What is the compulsion in using Stream Pipeline?**

* The compulsion in using Stream Pipeline is the presence of a terminal operation which helps in returning the final value and supports termination of the pipeline.

**3) Explain Predicate and Functions along with the code. What are the similarities?**

* Predicates and functions are both functional interfaces.
* **Predicate** is a **single argument** function that gives **the outcome as true or false**.
* Its code is <T>.Function is a single argument function that gives **outcomes in the form of an object**. Its code is <T, R>.

**4) What are the Core API classes for Java SE 8?**

* The Core API classes for Java SE 8 include LocalDate, LocalTime, and LocalDateTime.

5) **How is Collection API different from Stream API?**

The difference between Collection API and Stream API are:

|  |  |
| --- | --- |
| **Collection API** | **Stream API** |
| It is available since the introduction of Java 1.2 | It is made available after the introduction of Java SE 8 |
| It helps in storing Data which are set of objects. | It helps in computing data wherein computations of a set of data objects are made. |
| It can be used for storing a limited number of elements. | It can be used for storing an unlimited number of elements. |
| The construction of the Collection object is done Eagerly. | The construction of the Stream object is done Lazily. |

**6) How is Spliterator a different Iterator?**

A Spliterator in Java SE 8 is an iterator interface. The following are the differences between the spliterator and iterator:

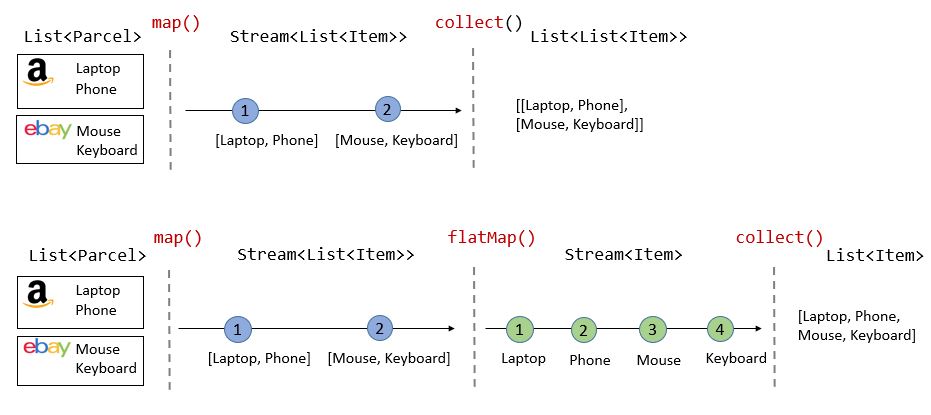
|  |  |
| --- | --- |
| **Spliterator** | **Iterator** |
| It is introduced along with Java SE 8 | It was introduced long ago with Java 1.2 |
| It is defined as a Spliterator iterator. | It is defined as a non-spilterator Iterator. |
| It can be used in Stream API. | It can be used in Collection API. |
| It helps to iterate streams in parallel and sequential order. | It helps in iterate collections only in sequential order. |
| The examples include tryAdvance() | The examples include, next(), hasNext(). |

#### 7) What is the major difference between Map and FlatMap stream operation?

The major difference between Map and FlatMap stream operation is that the former wraps its return value inside its ordinal type while the latter does not.

# **7.1)** [**What's the difference between map() and flatMap() methods in Java 8?**](https://stackoverflow.com/questions/26684562/whats-the-difference-between-map-and-flatmap-methods-in-java-8)

**One line answer: flatMap helps to flatten a Collection<Collection<T>> into a Collection<T>**. In the same way, it will also flatten an **Optional<Optional<T>>** into **Optional<T>**.



As you can see, with **map()** only:

* The intermediate type is Stream<List<Item>>
* The return type is List<List<Item>> and with **flatMap()**:
* The intermediate type is Stream<Item>
* The return type is List<Item>

This is the **test result** from the code used right below:

-------- Without flatMap() -------------------------------

collect() returns: [[Laptop, Phone], [Mouse, Keyboard]]

-------- With flatMap() ----------------------------------

collect() returns: [Laptop, Phone, Mouse, Keyboard]

## **.map is for***A -> B***mapping**

Stream.of("dog", "cat") // stream of 2 Strings

.map(s -> s.length()) // stream of 2 Integers: [3, 3]

it converts any item *A* to any item *B*

## **.flatMap is for***A -> Stream< B>***concatinating**

Stream.of("dog", "cat") // stream of 2 Strings

.flatMapToInt(s -> s.chars()) // stream of 6 ints: [d, o, g, c, a, t]

it --1 converts any item *A* into *Stream< B>*, then --2 concatenates all the streams into one (flat) stream

The function you pass to stream.map has to return one object. That means each object in the input stream results in exactly one object in the output stream.

The function you pass to stream.flatMap returns a stream for each object. That means the function can return any number of objects for each input object (including none). The resulting streams are then concatenated to one output stream.

**8) What is the similarity between Map and Flat map stream operation?**

Both the Map and FlatMap stream operations are intermediate stream operations that receive a function and also apply these functions to different elements of the stream.

9) **What are some examples of Intermediate Operations?**

Examples of Intermediate Operations include the following.

1. Limit(long n)
2. skip (long n)
3. Distinct ()
4. Filter (Predicate)
5. Map (Function)

#### 10)What are some examples of Terminal Operations?

Examples of terminal operations are:

1. Max
2. Min
3. AnyMatch
4. AllMatch
5. Reduce
6. ToArray
7. Count

11) **Finding certain strings without using Stream**

import java.util.ArrayList;

import java.util.List;

public class Example{

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

int count = 0;

for (String str : names) {

if (str.length() < 6)

count++;

}

System.out.println("There are "+count+" strings with length less than 6");

}

}

Output:

There are 3 strings with length less than 6

**Same example using Stream**

import java.util.ArrayList;

import java.util.List;

public class Example{

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

//Using Stream and Lambda expression

long count = names.stream().filter(str->str.length()<6).count();

System.out.println("There are "+count+" strings with length less than 6");

}

}

**Output:**

There are 3 strings with length less than 6

**12)What is the difference between these code?**

The output of both the examples are same, however there is a major difference between these examples if you consider the performance of the code.

In the first example, we are iterating the whole list to find the strings with length less than 6.

There is no parallelism in this code.

In the second example, the stream() method returns a stream of all the names, the filter() method returns another stream of names with length less than 6, the count() method reduces this stream to the result.

All these operations are happening parallelly which means we are able to parallelize the code with the help of streams.

Parallel execution of operations using stream is faster than sequential execution without using streams.

## **14)How to work with Stream in Java**

As we have seen in the above example, the working of stream can be explained in three stages: **1. Create a stream**

**2. Perform intermediate operations** on the initial stream to transform it into another stream and so on on further intermediate operations. In the above example, the filter() operation is intermediate operation, there can be more than one intermediate operations.

**3**. **Perform** **terminal operation** on the final stream to get the result. In the above example, the count() operation is terminal operation.

## **15)Java Stream Features**

1. Stream **does not store** the elements. it simply performs the aggregate operations(such as filter() and count() that we have seen in the above example) to get the desired stream of data.

2. **The aggregate operations** that we perform on the collection, array or any other data source **do not change** the data of the source, they simply return a new stream. For example the code we have seen above is filtering the strings with length less than 6 using the stream operations but it didn’t change the elements of the list.

3. **All the stream operations are lazy in** nature which means they are not executed until they are needed. For example, if we want to display only the first 2 elements of a list using stream, the stream operation would stop at the end of second iteration after displaying the second element of list

https://www.softwaretestinghelp.com/java-8-interview-questions/

**16)Java Stream Example 1: Iterating and displaying selected integers**

import java.util.stream.\*;

public class Example {

public static void main(String[] args){

Stream.iterate(1, count->count+1)

.filter(number->number%3==0)

.limit(6)

.forEach(System.out::println);

}

}

Output:

3

6

9

12

15

18

**17)Java Stream Example 2: Concatenating two streams**

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class Example {

public static void main(String[] args) {

//list 1

List<String> alphabets = Arrays.asList("A","B","C");

//list 2

List<String> names = Arrays.asList("Sansa","Jon","Arya");

//creating two streams from the two lists and concatenating them into one

Stream<String> opstream = Stream.concat(alphabets.stream(), names.stream());

//displaying the elements of the concatenated stream

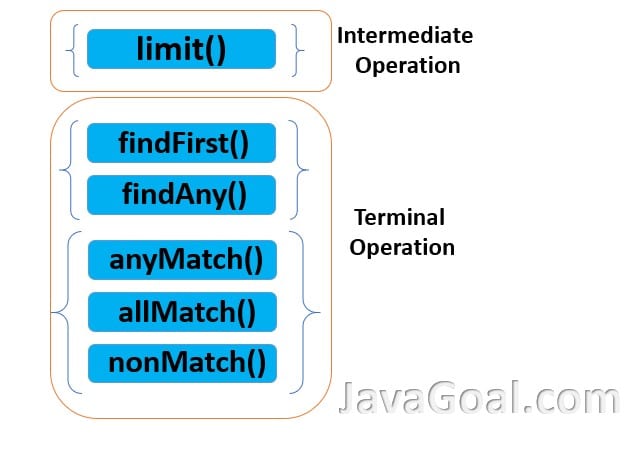
opstream.forEach(str->System.out.print(str+" "));

}

}

Output:

A B C Sansa Jon Arya



**Functional Interface**

1) What is the functional interface?

The functional interface is the interface which carries only one abstract method. The code example below explains this further.

program 1:

// Java program to demonstrate Implementation of

// functional interface using lambda expression

class Test

{

public static void main(String args[])

{

// lambda expression to create the object

new Thread(()->

).start();

}

}

**2) Is there any difference between the Functional interface and SAM interface?**

No, there is no difference between the functional interface and SAM interface. The SAM interface or Single Abstract Method interface is a kind of functional interface defined in the Java SE 8 API.

**3) Can we define a functional interface? If yes, how?**

Yes, we can define a functional interface. They can be defined using Java SE 8’s @Functionalinterface annotation to mark an interface as a functional interface.

**4) What are the guidelines that are needed to be followed for Functional Interface?**

There are several guidelines stated below which are needed to be followed for a functional interface. These are:

* The interface should be defined with only one abstract method.
* Not more than one abstract can be defined.
* Making use of @Functionalinterface annotation in the interface definition.
* The override of the Java.lang.object class’s method will not be considered as an abstract method.
* Any method can be used for defining a number.

**4) Which are the two most popular methods that can be used for defining any number in a Functional Interface?**

The two most popular methods used for defining any number in a functional interface include static methods and default methods.

**5) Is it mandatory to define a Functional Interface with @Functionalinterface annotation?**

No. it is not mandatory to define a functional interface with @Functionalinterface annotation.

**6) What is the compulsion in the Functional Interface definition?**

The compulsion in the functional interface definition is when the Java Compiler forces us to use one abstract inside the interface.

**7) What is the link between Lambda Expressions and Functional Interface?**

Using Lambda Expressions means that we are using a functional interface - they are interrelated. The lambda expressions are a part of the functional interface, which is a bigger platform carrying various other features and expressions.

**8)What are the common types of functional Interfaces in the standard library?**

The common types of functional interfaces in the standard library include the following:

1. Predicate
2. Function
3. Consumer
4. Supplier
5. BiFunction
6. BinaryOperator
7. UnaryOperator

* Function. Takes one argument and returns a result
* Consumer. Takes one argument and returns no result
* Supplier. Takes a not argument and returns a result
* Predicate. Takes one argument and returns a boolean
* BiFunction. Takes two arguments and returns a result
* BinaryOperator. It’s like a BiFunction, except it takes two arguments and returns a result, and they are all the same type
* UnaryOperator. It’s like a Function, but it takes a single argument and returns a result of the same type

## Java 8 Interview Questions and Answers

**What is Java 8?**  
Java 8 is a major release of the Java programming language that majorly changed the old style of programming by introducing several new features, such as, lambda expressions, functional interfaces, and streams.

**What is a lambda expression?**  
A lambda expression is a short way to express a function in Java. It allows us to write a function inline and pass it as an argument to another function as well.

**What is a functional interface?**  
A functional interface is an interface that contains exactly one abstract method. It is used to represent a single function contract.

**What is a stream in Java 8?**  
A stream is a sequence of elements that can be processed parallel or sequentially. Streams are a new addition to Java 8 and provide an easy way to work with collections.

**What are the benefits of using streams?**  
Streams provide a more concise and demonstrative way to work with collections. They also enable parallel processing of collections, which can lead to improved performance.

**What is the difference between a parallel stream and a sequential stream in Java 8?**  
A parallel stream allows for processing elements concurrently using multiple threads, while a sequential stream processes elements one-by-one in a single thread. To create a parallel stream, you can call the parallel() method on a stream.

**What is the difference between a stream and a collection in Java 8?**  
A stream is a sequence of elements that can be processed in parallel or sequentially, while a collection is a data structure that stores a group of elements. Additionally, Streams provide an easy way to work with Collections.

**What are the common terminal operations on streams?**  
Some common [terminal operations on streams](https://javatechonline.com/stream-api-in-java-8/#How_to_use_Stream_Terminal_Operations) include forEach(), reduce(), collect(), min(), max(), findFirst(), findAny() and count() etc.

**What is the difference between a terminal operation and an intermediate operation in a stream?**  
An intermediate operation on a stream returns a new stream, while a terminal operation consumes the stream and produces a result.

**What is the difference between a functional interface and a normal interface in Java 8?**  
A functional interface contains only one abstract method, while a normal interface can contain any number of abstract methods.

**What is a default method in Java 8?**  
A default method is a method with the ‘default’ keyword that is defined in an interface and has a default implementation. It can be overridden by an implementing class if needed.

**What is a static method in Java 8?**  
A static method is a method with the ‘static’ keyword that is defined in a class and can be called without creating an instance of the class.

**What is the syntax for a lambda expression in Java 8?**  
The syntax for a lambda expression in Java 8 is:

(parameter list) -> expression

**What is the difference between a lambda expression and an anonymous inner class in Java 8?**  
A lambda expression is a concise way to express a function, while an anonymous inner class is a way to create a class without giving it a name.

**What is a method reference in Java 8?**  
A method reference is a shorthand way to write a lambda expression that calls a method. Method references can be used to simplify code and improve readability by eliminating the need for a separate lambda expression. For example, instead of using “x -> Math.sqrt(x)”, we can use “Math::sqrt” as a method reference.

**What are the predefined functional interfaces in Java 8?**  
Some common [predefined functional interfaces in Java 8](https://javatechonline.com/predefined-functional-interfaces/) include Function, Predicate, Consumer, and Supplier.

**What is the Function interface in Java 8?**  
The Function interface in Java 8 is a functional interface that takes an argument and returns a result.

**What is the Predicate interface in Java 8?**  
The Predicate interface in Java 8 is a functional interface that takes an argument and returns a Boolean result.

**What is the Consumer interface in Java 8?**  
The Consumer interface in Java 8 is a functional interface that takes an argument and returns no result.

**What is the Supplier interface in Java 8?**  
The Supplier interface in Java 8 is a functional interface that takes no argument and returns a result of a specified type.

**What is the Optional class in Java 8?**  
The Optional class in Java 8 is a container object that may or may not contain a value. It is used to avoid null pointer exceptions.

**What is a CompletableFuture in Java 8?**  
A CompletableFuture in Java 8 is a class that represents a task that will be completed in the future. It can be used for asynchronous programming.

**What is the @FunctionalInterface annotation in Java 8?**  
The @FunctionalInterface annotation in Java 8 is used to indicate that an interface is a functional interface.

**What is the use of the map() method in Java 8 streams?**  
The map() method in Java 8 streams is used to transform each element in a stream into a new element.

**What is the use of the filter() method in Java 8 streams?**  
The filter() method in Java 8 streams is used to filter out elements from a stream based on a specified condition.

**What is the use of the reduce() method in Java 8 streams?**  
The reduce() method in Java 8 streams is used to combine all the elements in a stream into a single result.

**What is the use of the collect() method in Java 8 streams?**  
The collect() method in Java 8 streams is used to collect the elements in a stream into a specified data structure.

**What is the use of the flatMap() method in Java 8 streams?**  
The flatMap() method in Java 8 streams is used to flatten a stream of streams into a single stream.

**What is the use of the peek() method in Java 8 streams?**  
The peek() method in Java 8 streams is used to perform an operation on each element in a stream without modifying the stream.

**What is the use of the sorted() method in Java 8 streams?**  
The sorted() method in Java 8 streams is used to sort the elements in a stream.

**What is the use of the distinct() method in Java 8 streams?**  
The distinct() method in Java 8 streams is used to remove duplicate elements from a stream.

**What is the use of the skip() method in Java 8 streams?**  
The skip() method in Java 8 streams is used to skip a specified number of elements in a stream.

**What is the use of the limit() method in Java 8 streams?**  
The limit() method in Java 8 streams is used to limit the number of elements in a stream to a specified number.

**What is the use of the parallel() method in Java 8 streams?**  
The parallel() method in Java 8 streams is used to process the elements in a stream in parallel.

**What is the use of the sequential() method in Java 8 streams?**  
The sequential() method in Java 8 streams is used to process the elements in a stream in a sequential manner.

**What is the use of the allMatch() method in Java 8 streams?**  
The allMatch() method in Java 8 streams is used to check if all the elements in a stream satisfy a specified condition.

**What is the use of the anyMatch() method in Java 8 streams?**  
The anyMatch() method in Java 8 streams is used to check if any of the elements in a stream satisfy a specified condition.

**What is the use of the noneMatch() method in Java 8 streams?**  
The noneMatch() method in Java 8 streams is used to check if none of the elements in a stream satisfy a specified condition.

**What is the use of the findFirst() method in Java 8 streams?**  
The findFirst() method in Java 8 streams is used to return the first element in a stream that satisfies a specified condition.

**What is the use of the findAny() method in Java 8 streams?**  
The findAny() method in Java 8 streams is used to return any element in a stream that satisfies a specified condition.

**What is the use of the count() method in Java 8 streams?**  
The count() method in Java 8 streams is used to return the number of elements in a stream.

**What is the use of the toArray() method in Java 8 streams?**  
The toArray() method in Java 8 streams is used to convert a stream into an array.

**What is the use of the parallelStream() method in Java 8?**  
The parallelStream() method in Java 8 is used to create a parallel stream.

**What is the use of the sequentialStream() method in Java 8?**  
The sequentialStream() method in Java 8 is used to create a sequential stream.

**What is the use of the flatMapToInt() method in Java 8 streams?**  
The flatMapToInt() method in Java 8 streams is used to flatten a stream of streams into a single stream of integers.

**What is the use of the flatMapToLong() method in Java 8 streams?**  
The flatMapToLong() method in Java 8 streams is used to flatten a stream of streams into a single stream of longs.

**What is the use of the flatMapToDouble() method in Java 8 streams?**  
The flatMapToDouble() method in Java 8 streams is used to flatten a stream of streams into a single stream of doubles.

**Why was lambda expression introduced in Java 8?**  
Lambda expressions were introduced in Java 8 to provide a concise and functional way of implementing interfaces with a single abstract method, also known as functional interfaces. Lambda expressions allow developers to write code that is more concise and expressive.

**Why was the purpose of forEach() method added to the Iterable interface in Java 8?**  
The forEach() method was added to the Iterable interface in Java 8 to provide a simple and concise way of iterating over collections. The forEach() method allows developers to write code that is more readable and expressive.

**Why was the default method introduced in Java 8 interfaces?**  
The default method was introduced in Java 8 interfaces to provide a way to add new methods to existing interfaces without breaking backwards compatibility. Default methods provide a way to extend the functionality of interfaces in a safe way.

**Why was the Date and Time API introduced in Java 8?**  
The Date and Time API was introduced in Java 8 to provide a more robust and flexible way of handling dates and times. The new API allows developers to handle dates and times in an easier way.

**Why was the reduce() method added to the Stream interface in Java 8?**  
The reduce() method was added to the Stream interface in Java 8 to provide a way of reducing a collection of data to a single value. The reduce() method allows developers to perform complex operations on data in a simple and efficient way.

**Why was the parallel processing introduced in Java 8?**  
Parallel processing was introduced in Java 8 to provide a way to take advantage of multi-core processors. Parallel processing allows developers to perform complex operations on data in a faster and more efficient way.

**Why is the purpose of Spliterator interface in Java 8?**  
The Spliterator interface was introduced in Java 8 to provide a way to split a collection of data into smaller parts. The Spliterator interface allows developers to process large collections of data in a more efficient way.

**Why is the purpose of  BiFunction interface introduced in Java 8?**  
The BiFunction interface was introduced in Java 8 to allow a way of passing two arguments at a time to a function and returning a result. The BiFunction interface allows developers to write code that is more expressive and concise.

**Why was the CompletableFuture class introduced in Java 8?**  
The CompletableFuture class was introduced in Java 8 to provide a way of performing asynchronous operations. The CompletableFuture class allows developers to write code that is more responsive and efficient.

**Why was the groupingBy() method added to the Collectors class in Java 8?**  
The groupingBy() method was added to the Collectors class in Java 8 to provide a way of grouping elements based on a certain criteria. The groupingBy() method allows developers to process collections of data in a more flexible and efficient way.

**Why was the toArray() method added to the Stream interface in Java 8?**  
The toArray() method was added to the Stream interface in Java 8 to provide a way of converting a stream into an array. The toArray() method allows developers to process collections of data in a more flexible and efficient way.

**Why is the java.util.function package important in Java 8?**  
The java.util.Function package is important in Java 8 because it offers a flavor of functional programming as a set of functional interfaces that can be used with lambda expressions. The java.util.function package allows developers to write code that is more expressive and concise.

**Why was the peek() method added to the Stream API in Java 8?**  
The peek() method was added to the Stream API in Java 8 to allow developers to debug and understand their code more easily.

**Why was the Optional class introduced in Java 8?**  
The Optional class was introduced in Java 8 to provide a way of handling null values in a more concise and expressive way. The Optional class allows developers to write code that is more robust and bug-free.

**Why was the trySplit() method added to the Spliterator interface in Java 8?**  
The trySplit() method was added to the [Spliterator](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html) interface in Java 8 to provide a way of splitting a collection of data into two separate streams. The trySplit() method allows developers to write code that is more flexible and efficient.

**Why was the min() and max() methods added to the Stream interface in Java 8?**  
The min() and max() methods were added to the Stream interface in Java 8 to provide an easy way of finding the minimum and maximum values in a collection of data.

**Why is the purpose of getOrDefault() method added to the Map interface in Java 8?**  
The getOrDefault() method was added to the Map interface in Java 8 to provide a way of getting a value from a map with a default value if the key is not found.

**Why was the computeIfAbsent() method added to the Map interface in Java 8?**  
The computeIfAbsent() method was added to the Map interface in Java 8 to provide a way of getting a value from a map and computing a new value if the key is not found.

**Why was the takeWhile() method added to the Stream interface in Java 8?**  
The takeWhile() method was added to the Stream interface in Java 8 to provide a way of selecting elements from a stream until a certain condition is met.

**Why was the dropWhile() method added to the Stream interface in Java 8?**  
The dropWhile() method was added to the Stream interface in Java 8 to provide a way of selecting elements from a stream after a certain condition is met.

**Why was the or() method added to the Predicate interface in Java 8?**  
The or() method was added to the Predicate interface in Java 8 to provide a way of combining multiple predicates into a single predicate. It acts as a short-circuiting logical OR of this predicate and another.

**Why was the and() method added to the Predicate interface in Java 8?**  
The and() method was added to the Predicate interface in Java 8 to provide a way of combining multiple predicates into a single predicate. It acts as a short-circuiting logical AND of this predicate and another.

**Why was the asDoubleStream() method added to the IntStream interface in Java 8?**  
The asDoubleStream() method was added to the IntStream interface in Java 8 to provide a way of converting an IntStream to a DoubleStream.

**Why was the asLongStream() method added to the IntStream interface in Java 8?**  
The asLongStream() method was added to the IntStream interface in Java 8 to provide a way of converting an IntStream to a LongStream.

**Why was the ofNullable() method added to the Optional class in Java 8?**  
The ofNullable() method was added to the Optional class in Java 8 to provide a way of creating an Optional object with a null value. The ofNullable() method is used to get an instance of the Optional class with a specified value. If the value is null, then an empty Optional object is returned.

**Why was the flatMapToInt() method added to the Stream interface in Java 8?**  
The flatMapToInt() method was added to the Stream interface in Java 8 to provide a way of flattening a stream of objects to an IntStream.

**Why was the toMap() method added to the Collectors class in Java 8?**  
The toMap() method was added to the Collectors class in Java 8 to provide a way of collecting a stream of objects to a Map object.

**Why was the of() method added to the Optional class in Java 8?**  
The of() method was added to the Optional class in Java 8 to provide a way of creating an Optional object. It will return an Optional object containing the given value if the value is non-null, or an empty Optional object if the value is null.

♥ If you want to attempt multiple choice questions on Java 8 Features, kindly visit a separate article on ‘[MCQs on Java 8 Features](https://javatechonline.com/mcq-on-java-8-features/)‘ with answers and explanations.