**What New Features Were Added in Java 8?**

<https://javahungry.blogspot.com/2018/10/java-8-interview-questions-and-answers.html>

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

### ****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.

**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

### ****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:

|  |
| --- |
| 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.

# **Stream In Java**

Introduced in Java 8, the 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.  
The features of Java stream are –

* 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.

Different Operations On Streams-  
**Intermediate Operations:**

1. **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());
2. **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());
3. **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.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

**Important Points/Observations:**

1. A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
2. Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

# **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:

|  |  |
| --- | --- |
| 1 | (o) -> o.toString(); |

can become:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1 | String::new; |

Static method reference:

|  |  |
| --- | --- |
| 1 | String::valueOf; |

Bound instance method reference:

|  |  |
| --- | --- |
| 1 | str::toString; |

Unbound instance method reference:

|  |  |
| --- | --- |
| 1 | String::toString; |

# **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.

You can read a dedicated article about Optional [here](https://www.baeldung.com/java-optional).

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:

|  |  |
| --- | --- |
|  | int min1 = Arrays.stream(new int[]{1, 2, 3, 4, 5})    .min()    .orElse(0);  assertEquals(1, min1);    int min2 = Arrays.stream(new int[]{})    .min()    .orElse(0);  assertEquals(0, min2);  // Java program without Optional Class  public class OptionalDemo{      public static void main(String[] args) {          String[] words = new String[10];          String word = words[5].toLowerCase();          System.out.print(word);      }  }  Exception in thread "main" java.lang.NullPointerException  // Java program with Optional Class  import java.util.Optional;  public class OptionalDemo{      public static void main(String[] args) {          String[] words = new String[10];          Optional<String> checkNull =                        Optional.ofNullable(words[5]);          if (checkNull.isPresent()) {              String word = words[5].toLowerCase();              System.out.print(word);          } else              System.out.println("word is null");      }  }  Output :  word is null |

**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

### ****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.

### ****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.

### 