Java Interview questions

1. Why you use stream?

Streams are Java’s way of integrating functional programming with its object-oriented style. There are a lot of benefits to using streams in Java, such as the ability to write functions at a more abstract level which can reduce code bugs, compact functions into fewer and more readable lines of code, and the ease they offer for parallelization. Java streams are fairly well known but not everyone knows how to take full advantage of their benefits, including the finer points of making streams, intermediate operations, and terminal operations. In this blog we’re going to use a simple example to explain how Java streams work, which will result in less verbose, more intuitive, and less error prone code.

1. Difference between Stream and Collection?

Streams differ from collections in several ways:

* + **No storage**. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.
  + **Functional in nature**. An operation on a stream produces a result, but does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
  + **Laziness-seeking**. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, "find the first String with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.
  + **Possibly unbounded**. While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.
  + **Consumable**. The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

In contrast, a [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) is a *container* of objects (elements). You can't get (retrieve) an object from a collection unless the object was previously added to the collection.

1. Difference between Stream and Parallel stream?

The only difference between the serial and parallel versions of this example is the creation of the initial stream, using "parallelStream()" instead of "stream()". When the terminal operation is initiated, the stream pipeline is executed sequentially or in parallel depending on the orientation of the stream on which it is invoked. Whether a stream will execute in serial or parallel can be determined with the isParallel() method,

1. What is stream pipeline?

* Stream operations are divided into *intermediate* and *terminal* operations, and are combined to form *stream pipelines*. A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.
* Intermediate operations return a new stream. They are always *lazy*; executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.
* Terminal operations, such as Stream.forEach or IntStream.sum, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used; if you need to traverse the same data source again, you must return to the data source to get a new stream. In almost all cases, terminal operations are *eager*, completing their traversal of the data source and processing of the pipeline before returning. Only the terminal operations iterator() and spliterator() are not; these are provided as an "escape hatch" to enable arbitrary client-controlled pipeline traversals in the event that the existing operations are not sufficient to the task.

1. What is functional interface and example?

A functional interface is an interface with only one abstract method. This means that the interface implementation will only represent one behavior. Examples of a functional interface in Java are:

* java.lang.Runnable
* java.util.Comparator
* java.util.concurrent.Callable
* java.io.FileFilter

An important point to remember is that the functional interface can have a number of default methods but only one abstract method.

1. Why default keyword and its benefit?

Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.

1. Difference between aggregate operation and iterator.
2. Difference between Map and Flat Map.
3. What is use of Optional class?
4. Give example of inbuilt functional interface ?
5. Usage of Consumer, Supplier , Predicate , Function interface
6. What is string joiner and its usage?
7. What is lambda expression and method referencing?
8. Difference between Function and By Function.
9. How many ways to obtain a stream?
10. What is the difference between PermGen and MetaSpace?

Some of the important Java 8 features are:-

A diagram of a program

Description automatically generated

1. forEach() method in Iterable interface
2. default and static methods in Interfaces
3. Functional Interfaces and Lambda Expressions
4. Java Stream API for Bulk Data Operations on Collections
5. Java Time API
6. Collection API improvements
7. Concurrency API improvements
8. Java IO improvements

Some of the important Java 11 features are:-

1. Running Java File with single command
2. New utility methods in String class
3. Local-Variable Syntax for Lambda Parameters
4. Nested Based Access Control
5. JEP 321: HTTP Client
6. Reading/Writing Strings to and from the Files
7. JEP 328: Flight Recorder

Some of the important Java 17 features are:-

1. Deprecate the Applet API for Removal
2. Pattern matching for Switch(Preview)

## Sealed Classes Best Java Frameworks For Microservices:-

## Spring Boot

## Quarkus

## Micronaut

## 