Welcome

Java Streams and Reflection

Prakash Badhe

prakash.badhe@vishwasoft.in

Copyright Notice

Immutability in Java

- An immutable object is an object that will not change its internal state after creation.
- The String and Wrapper class objects are immutable.
- The immutable objects *can be shared between threads without synchronization*., they are always **thread safe.**
- The immutable objects increases the scalability of application.

The final vs. immutable

- For a variable is declared with final keyword, it’s value can’t be modified, essentially, a constant.
- With final can't change the object's reference to point to another reference or another object, but you can still mutate its state (using setter methods etc.)
- Immutability means unchanging over time or unable to be changed with **immutable** the object's actual value can't be changed, but you can change its reference to another one.

The state of immutable

- The final ensures that the address of the object remains the same whereas the Immutable suggests that we can’t change the state of the object once created.

Immutable class

- Class must be declared as final (So that child classes can’t be created)
- Data members in the class must be declared as final (So that the value of it after object creation cannot be changed)
- Create a private constructor and a factory to create instances of the immutable class.
- Getter method for all the variables in it.
- No setters(To not have the option to change the value of the instance variable)

Functional Programming

- Functional programming is a paradigm that allows programming using expressions i.e. declaring functions, passing functions as arguments and using functions as statements.
- The functions are first class objects.

Lambda with Java8

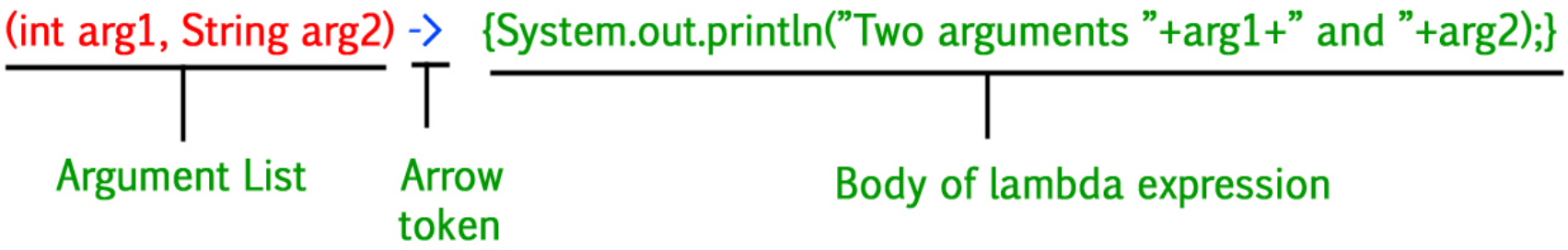
- Lambda expressions express instances of functional interfaces.
- An interface with single abstract method is called functional interface. An example is java.lang.Runnable.
- The lambda expressions implement the only abstract function and therefore implement functional interfaces.

Lambda implements functional

- Lambda expressions basically express instances of functional interfaces (An interface with single abstract method is called functional interface. E.g.java.lang.Runnable).
- The lambda expressions implement the only abstract function and therefore implement functional interfaces.

Lambda with functions

- Enable to treat functionality as a method argument, or code as data.
- A function that can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and executed on demand.
- Lambda is a method without name.(anonymous function)



Lambda with parameters

- **Zero parameter**
  () -> System.out.println("Zero parameters with lambda");
- **One parameter:**
- (p) -> System.out.println("One parameter with lambda: " + p);
- **Multiple parameters**
- (p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

Lambda Specific

- It can't extend abstract and concrete class.
- It can implement an interface which contains a single abstract method.
- It does not allow declaration of instance variables, whether the variables declared simply act as local variables.
- Lambda Expression can't be instantiated.
- Inside Lambda Expression, "this" always refers to current outer class object that is, enclosing class object.
- best choice if to handle interface.

Lambda Details

- At the time of compilation, no separate .class file will be generated. It simply convert it into private method outer class.
- It resides in a permanent memory of JVM.

Streams with Java

- Stream is a continuous flow (of data)
- To process collections of objects, Stream API is defined with java8.
- Streams are wrappers around a data source, allowing to operate with that data source and making bulk processing convenient and fast.
- A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result..

Stream Types

- Streams be created from various data sources, especially collections.
- Lists and Sets support new methods stream() and parallelStream() to either create a sequential or a parallel stream.
- Parallel streams are capable of operating on multiple threads.

Stream Support

- Java 8 supports special kinds of streams for working with the primitive data types int, long and double.
- The IntStream, LongStream and DoubleStream.
- Primitive streams use specialized lambda expressions, e.g. IntFunction instead of Function or IntPredicate instead of Predicate.
- The primitive streams support the additional terminal aggregate operations sum() and average()

Streams on data

- Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels
- The intermediate operations with Streams returns a stream as a result, hence various intermediate operations can be pipelined.
- Terminal operations of Streams mark the end of the stream and return the result.

Stream Creation

- Creation of an empty stream:
- Stream<String> streamEmpty = Stream.empty();
- Stream created of any type of *Collection*
- Collection<String> collection = Arrays.asList("a", "b", "c");
- Stream<String> streamOfCollection = collection.stream();
- **Stream of Array**
- Stream<String> streamOfArray = Stream.of("a", "b", "c")

Stream Builder

- **When builder is used the desired type should be additionally specified in the right part of the statement,** otherwise the *build()* method will create an instance of the *Stream<Object>*
- Stream<String> streamOfStrings =
- Stream.<String>builder().add("a").add("b").add("c").build();

Stream Generate

- The ***generate()*** method accepts a *Supplier<T>* for element generation.
- As the resulting stream is infinite, should specify the desired size or the *generate()* method will work until it reaches the memory limit.
- Create a sequence of ten strings with the value – *"element"*.
- Stream<String> streamGenerated =
- Stream.generate(() -> "element").limit(10);

Stream Iterate

- Create an infinite stream is by using the ***iterate()*** method
- Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);
- The first element of the resulting stream is a first parameter of the *iterate()* method. For creating every following element the specified function is applied to the previous element.
- Above the second element is 42.

Stream of Primitives

- Streams out of three primitive types: *int, long* and *double.*
- Three new special interfaces defined: ***IntStream, LongStream, DoubleStream.***
- Using the new interfaces alleviates unnecessary auto-boxing allows increased productivity:

Stream of *String*

- *A String* can also be used as a source for creating a stream.
- With the help of the *chars()* method of the *String* class.
- IntStream streamOfChars = "abc".chars();
- Break a *String* into sub-strings according to specified *RegEx*
- Stream<String> streamOfString =
- Pattern.compile(", ").splitAsStream("a, b, c");

Stream of File

- Java NIO class *Files* allows to generate a *Stream<String>* of a text file through the *lines()* method. Every line of the text becomes an element of the stream
- *Path path = Paths.get("C:\\file.txt");*
- *Stream<String> streamOfStrings = Files.lines(path);*
- *Stream<String> streamWithCharset =*
- *Files.lines(path, Charset.forName("UTF-8"));*

Stream operations

- Map: returns a stream consisting of the results of applying the given function to the elements of this stream.
- **filter:** The filter method is to select elements as per the Predicate passed as argument.
- **sorted:** The sorted is used to sort the stream.

Lazy Streams

- The stream on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.
- All intermediate operations are lazy, so they're not executed until a result of a processing is actually needed and decided by terminal operation.

Stream terminal operations

- **collect:** The collect method is to return the result of the intermediate operations performed on the stream.
- for**Each:** to iterate through every element of the stream.
- **reduce:** to reduce the elements of a stream to a single value. It takes a BinaryOperator as a parameter.

Stream structure

- 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.
- Stream is used to compute elements as per the pipelined methods without altering the original value of the object

Stream Parameters

- Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation.
- Most of those operations must be both *non-interfering* and *stateless.*
- A non-interfering function does not modify the underlying data source of the stream

Stream Operations

- The object streams support the special mapping operations mapToInt(), mapToLong() and mapToDouble
- The intermediate operations will only be executed when a terminal operation is present.

Reduction Operations

- A reduction operation (also called as fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.
- The reduction operations like *findFirst()*, *min()* and *max*().

Stream Pipeline

- To perform a sequence of operations over the elements of the data source and aggregate their results, three parts are needed –
  the **source**, **intermediate operation(s)** and
  a **terminal operation.**
- Intermediate operations return a new modified stream.
- *To create a new stream of the existing one without few elements the skip() method should be used*
- *Stream<String> onceModifiedStream =*
- *Stream.of("abcd", "bbcd", "cbcd").skip(1);*

Parallel Streams

- Achieve parallelism in a functional style.
- The API allows creating parallel streams, which perform operations in a parallel mode.
- When the source of a stream is a *Collection* or an *array* it can be achieved with the help of the ***parallelStream()***
- Stream API automatically uses the *ForkJoin* framework to execute operations in parallel.

Avoid blocking in parallel

- With streams in parallel mode, avoid blocking operations and use parallel mode when tasks need the similar amount of time to execute (if one task lasts much longer than the other, it can slow down the complete app's workflow).
- The stream in parallel mode can be converted back to the sequential mode by using the *sequential()* method
- IntStream intStreamSequential = intStreamParallel.sequential();
- boolean isParallel = intStreamSequential.isParallel();

Java reflection API

- Reflection is an API used to examine or modify the behavior of methods, classes, variables, interfaces etc. at runtime.
- Provided under java.lang.reflect package.
- ReflectionPI gives information about the class to which an object belongs and also the methods of that class which can be executed by using the object.

Reflection Components

- Class
- Constructor
- Method
- Field
- Annotations
- Arrays
- Modules
- Dynamic Proxy and Generic Types

Reflection loses the privacy of class

- With reflection methods can be invoked at runtime irrespective of the access specifier used with them.
- With reflection we can **access the private variables and methods** of a class with the help of its class object and invoke the method by using the object.
- C**lass.getDeclaredField(FieldName) :** Used to get the private field. Returns an object of type Field for specified field name.
  Field.setAccessible(true) : Allows to access the field irrespective of the access modifier used with the field.

Reflection Usage

- **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Debugging and testing tools**: Debuggers use the property of reflection to examine private members on classes.
- High level frameworks internally use reflection to analyze the classes, objects and manage dependency injection on private variables.
- Examples @Resource, @Inject,@Autowired etc.

Reflection in Production Code..?

- Avoid it.
- **Performance Overhead:** Reflective operations have slower performance and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.