Java 8

Java 8

- JAVA 8 is a major feature release of JAVA programming language development.

Java 8 – New Features

- **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.

Java 8 – New Features

- **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.

Java 8 - Lambda Expressions

- Lambda expressions are introduced in Java 8 Lambda expression facilitates functional programming, and simplifies the development a lot.
- A lambda expression is characterized by the following syntax.

```
(argument-list) -> {body}
```

Java 8 - Lambda Expressions

(argument-list) -> {body}

- Java lambda expression is consisted of three components.
- **Argument-list:** It can be empty or non-empty as well.
- **Arrow-token:** It is used to link arguments-list and body of expression.
- **Body:** It contains expressions and statements for lambda expression.

Java 8 - Lambda Expressions

- **No Parameter Syntax**

```
() -> {
//Body of no parameter lambda
}
```

- **One Parameter Syntax**

```
(p1) -> {
//Body of single parameter lambda
}
```

- **Two Parameter Syntax**

```
(p1,p2) -> {
//Body of multiple parameter lambda
}
```

Java 8 - Lambda Expressions

- 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.

Java Functional Interfaces

- An Interface that contains exactly one abstract method is known as functional interface.
- It can have any number of default, static methods but can contain only one abstract method.
- It can also declare methods of object class.
- Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces.

Java Default Methods

- Java provides a facility to create default methods inside the interface.
- Methods which are defined inside the interface and tagged with default are known as default methods.
- These methods are non-abstract methods.

Method References in Java 8

- Method reference is a shorthand notation of a lambda expression to call a method.
- For example:
  - If your lambda expression is like this:

```
str -> System.out.println(str)
```

  - then you can replace it with a method reference like this:

```
System.out::println
```

  - The :: operator is used in method reference to separate the class or object from the method name

Java - New Date/Time API

- With Java 8, a new Date-Time API is introduced to cover the following drawbacks of old date-time API.
- **Not thread safe**
- **Poor design**
- **Difficult time zone handling**

Java - New Date/Time API

- Java 8 introduces a new date-time API under the package java.time. Following are some of the important classes introduced in java.time package.
- **Local** − Simplified date-time API with no complexity of timezone handling.
- **Zoned** − Specialized date-time API to deal with various timezones.

Java - Stream

- Java provides a new additional package in Java 8 called java.util.stream.
- This package consists of classes, interfaces and enum to allows functional-style operations on the elements.
- You can use stream to filter, collect, print, and convert from one data structure to other etc.

Java - Stream

- Java 8 Streams should not be confused with Java I/O streams (ex: *FileInputStream* etc); these have very little to do with each other.
- Simply put, streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.
- **A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.**
- This functionality – *java.util.stream* – supports functional-style operations on streams of elements, such as map-reduce transformations on collections.

Java Stream Operations

- some common usages and operations we can perform on and with the help of the stream support in the language

forEach

- *forEach()* is simplest and most common operation; it loops over the stream elements, calling the supplied function on each element.
- ***forEach()* is a terminal operation**, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used.

map

- *map()* produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

collect

- *collect()* performs mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in the *Stream* instance.
- The strategy for this operation is provided via the *Collector* interface implementation.

filter

- *filter()*; produces a new stream that contains elements of the original stream that pass a given test (specified by a Predicate).

reduce()

- The Stream.reduce() method is a reduction operation. A reduction operation takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.

Lazy Evaluation

- One of the most important characteristics of Java streams is that they allow for significant optimizations through lazy evaluations.
- **Computation 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.**

Java 8 Optional Class

- Java 8, have a newly introduced Optional class in java.util package.
- This class is introduced to avoid NullPointerException that we frequently encounters if we do not perform null checks in our code.
- Using this class we can easily check whether a variable has null value or not and by doing this we can avoid the NullPointerException.

Optional Class – Important Methods

- **Optional.ofNullable()** method of the Optional class, returns a Non-empty Optional if the given object has a value, otherwise it returns an empty Optional.
- **isPresent()** We can check whether the returned Optional value is empty or non-empty using the isPresent() method.
- **ifPresent()** There is another method present in the Optional class, which only executes if the given Optional object is non-empty.
- two methods **orElse()** and **orElseGet()** returns the value of Optional Object if it is no empty, if the object is empty then it returns the default value passed to this method as an argument