Java 8 Features

- Lambda Expressions
- Functional Interfaces
- Default methods in interfaces
- static methods in interfaces
- predicate
- function
- consumer
- Stream API
- Date and Time API

Functional Interfaces

A functional interface is an interface that **contains only one abstract method**. They can have only one functionality to exhibit. From Java 8 onwards**, lambda expressions** can be used to represent the **instance** of a **functional interface**. A functional interface can have any number of **default methods**. ***Runnable***, ***ActionListener***, ***Comparable*** are some of the examples of functional interfaces.

@FunctionalInterface annotation is used to ensure that the functional interface can't have **more than one abstract method**. In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message. However, it is not mandatory to use this annotation.

Predicate

Function

Lambda Expressions

Lambda expressions basically express instances of functional interfaces (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda expressions implement the only abstract function and therefore implement functional interfaces
lambda expressions are added in Java 8 and provide below functionalities.

- 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.

Default methods in interfaces

Java 8 introduces the "Default Method" or (Defender methods) feature, which allows the developer to add new methods to the interfaces without breaking their existing implementation. It provides the flexibility to allow interface to define implementation which will use as the default in a situation where a concrete class fails to provide an implementation for that method.

Why the Default Method?

Reengineering an existing JDK framework is always very complex. Modifying one interface in a JDK framework breaks all classes that extend the interface, which means that adding any new method could break millions of lines of code. Therefore, *default methods* have introduced as a mechanism to extend interfaces in a backward-compatible way.

Default methods can be provided to an interface without affecting implementing classes as it includes an implementation. If each added method in an interface is defined with implementation, then no implementing class is affected. An implementing class can override the default implementation provided by the interface.

static methods in interfaces

Static Methods in **Interface** are those methods, which are defined in the interface with the keyword static. Unlike other methods in Interface, these static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.
Similar to Default Method in Interface, the static method in an interface can be defined in the interface, but these methods cannot be overridden in Implementation Classes. To use a static method, Interface name should be instantiated with it, as it is a part of the Interface only.

Stream API

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.

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.