Lambdas

FEATURES OF LAMBDAS:

1. Enables functional programming
2. Readable and concise code
3. Easier-to-use APIs and libraries
4. Enables support for parallel processing

FUNCTIONAL PROGRAMMING VS OBJECT-ORIENTED PROGRAMMING:

You can do anything with object oriented programming. So, Functional programming does same as OOP, but it allows you to write readable and understandable code.

Functions as Values:

Assign a function i.e block of code to variable.
variable_name = (arguments) -> { };

Inline values:

String name = “foo”;
double pi = 3.14;

Notes:

1. The body of a lambda expression can contain zero, one or more statements.
2. When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
3. When there are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

Functional interface

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.

T: denotes the type of the input argument to the operation
R: denotes the type of the output argument to the operation

Before **Java 8, we had to create anonymous inner class objects or implement these interfaces.**

@FunctionalInterface Annotation
@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.

java.util.function Package:
The java.util.function package in Java 8 contains many **builtin functional interfaces** like-

1. Predicate: The Predicate interface has an abstract method test which gives a Boolean value as a result for the specified argument. Its prototype is
Predicate<T> name = (variableName) -> {condition}; // Return Boolean
name.test(Input of type T);
We can also use AND and OR operator in the predicates like:
name.**and**(name2).**test**(Input of type T);
name.**or**(name2).**test**(Input of type T);
Note: We can only use AND and OR with predicate with predicate / bipredicate with bipredicate.

2. BinaryOperator: The BinaryOperator interface has an abstract method apply which takes two argument and returns a result of same type. Its prototype is
BinaryOperator<T> name = (arg1, arg2) -> {}; // Return output of same time
For functional chaining in Binary Operator:
BiFunction<T,U,R> res = sum.andThen(multiply);
res.apply(20, 30);
There is also minBy() and maxBy() functionality in BinaryOperator:
Comparator<Salary> salaryComparator = Comparator.comparing(Salary::getVal);
Collectors.reducing(BinaryOperator.maxBy(salaryComparator));

3. Function: The Function interface has an abstract method apply which takes argument of type T and returns a result of type R. Its prototype is
Function<T, R> name = arg1 -> {}; // Return output of R type
BiFunction<T, U, R> biName = (arg1, arg2) -> {}; // Return output of R type
// Use Function with **apply**(Input)
name.apply(i);
// **Function Chaining**
Function<T, T> temp = name.**andThen**(name2);
Note: We can only use Function chaining with Function -> Function or BiFunction -> BiFunction

4. Consumer: It represents a function which takes in one argument and produces a result. However these kind of functions don't return any value.
Consumer<T> name = arg1 -> {}; // Return type void
BiConsumer<T, U> name = (arg1, arg2) -> {}; // Return type void

Note: We can also use functional changing with Consumer and BiConsumer

5. Supplier: It represents a function which doesn't take any input and produces a result of type T. Its prototype is

```
static Supplier<T> randomInt = () -> {return new Random().nextInt(1000);};
randomInt.get();
```

Note: Functional chaining not possible with supplier

Important Points/Observations:

- A functional interface has only one abstract method but it can have multiple default methods.
- @FunctionalInterface annotation is used to ensure an interface can't have more than one abstract method. The use of this annotation is optional.
- The java.util.function package contains many builtin functional interfaces in Java 8.

Optionals

The purpose of Optional is not to replace every single null reference in your code base but rather to help you design better APIs in which, just by reading the signature of a method, users can tell whether to expect an optional value and deal with it appropriately.

What is Optional trying to solve?

Optional is an attempt to reduce the number of null pointer exceptions in Java systems, by adding the possibility to build more expressive APIs considering that sometimes return values are missing. If Optional was there since the beginning, today most libraries and applications would likely deal better with missing return values, reducing the number of null pointer exceptions and the overall number of bugs in general.

By using Optional, user is forced to think about the exceptional case. Besides the increase in readability that comes from giving null a name, the biggest advantage of Optional is its idiot-proof-ness. It forces you to actively think about the absent case if you want your program to compile at all, since you have to actively unwrap the Optional and address that failure cases.

What is Optional not trying to solve?

Optional is not meant to be a mechanism to avoid all types of null pointers. e.g. The mandatory input parameters of methods and constructors will still have to be tested.

Like when using null, Optional does not help with conveying the meaning of an absent value. So the caller of the method will still have to check the javadoc of the API for understanding the meaning of the absent Optional, in order to deal with it properly.

Please note that Optional is not meant to be used in these below contexts, as possibly it won't buy us anything:

- in the domain model layer (it's not serializable)
- in DTOs (it's not serializable)

- in input parameters of methods
- in constructor parameters

How should Optional be used?
Optional should be used almost all the time as the return type of functions that might not return a value.
This is a quote from OpenJDK mailing list:
The JSR-335 EG felt fairly strongly that Optional should not be on any more than needed to support the optional-return idiom only.
Someone suggested maybe even renaming it to “OptionalReturn“.
This essentially means that Optional should be used as the return type of certain service, repository or utility methods only where they truly serve the purpose.

Stream API

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:
- **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());
  ```
- **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());
  ```
- **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:
- **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());
```

- **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));
  ```
- **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 .

Date Time API

All notes mentioned in the code.

https://github.com/aman9824/java8