**Functional programming in Java 8**

Java 8 supports functional programming. The features that support that are:

1. Lambda Expressions
2. Streams API
3. Optional Object

Let’s examine the following code to understand the concepts:

Map <String, List<String>> phoneNumbers = new HashMap <String, List<String>>();

phoneNumbers.put("John Lawson", Arrays.asList("3232312323", "8933555472"));

phoneNumbers.put("Mary Jane", Arrays.asList("12323344", "492648333"));

phoneNumbers.put("Mary Lou", Arrays.asList("77323344", "938448333"));

Map <String, List<String>> filteredNumbers = phoneNumbers.entrySet().stream()

.filter(x - > x.getKey().contains("Mary"))

.collect(Collectors.toMap(p - > p.getKey(), p - > p.getValue()));

filteredNumbers.forEach((key, value) - > {

System.out.println("Name: " + key + ": ");

value.forEach(System.out::println);

});

* Imperative programming:

Much like imperative mood in the speech, it tells the program to perform some command. It also changes the state of the program (i.e. variable values are changed before and after the execution)

* Functional programming:

It is a subset of declarative programming paradigm. It also has commands but the commands are treated more like mathematical functions. Their task is to take some input, perform some action and return a result. The essential part here is that they *do not modify* the state of the program. The input variable(s) remain unchanged, and the returned result is always a new variable.

Functional programming is a paradigm that favors stateless operations and avoids modifications to the programs state

Methods as used in the middle section:

* entrySet: To get a set of entries each consisting of a key and a value
* stream: To get a sequence of elements supporting sequential and parallel aggregate operations
* filter: To filter the items in the stream with some criteria. The criteria are expressed as x - > x.getKey().contains(“Mary”)

This is called a lambda expression. This concept is also introduced in Java 8. Here it acts like a predicate, a Boolean function. Its goal is to evaluate the filtering criteria and tell whether each specific item in the collection (stream) should be kept or removed.

* collect: To take the elements of the stream (java.util.Map.Entry) and converts them back into a regular collection.

As you can see, the methods are chained, and they all belong to the Stream interface, so this suggests that each of them returns the stream to be accessible for subsequent methods in the invocation process.

Using forEach loop on collection:

* The last part of the above code segment illustrates the use of forEach loop in Java. This feature also introduced in Java 8.
* It performs some action on each element of the underlying collection
* The actual operation to be performed here is defined by a lambda expression

**Lambda expressions**

* Lambda expressions are defined by:
  + Parameters - > expression body
  + For example: Comparator comparator = (a, b) - > a.compareTo(b);
  + It takes a parameter or parameters and does stuff with them

* Type declaration is optional. Compiler can figure out the types from the input parameters
* Brackets not needed around a single parameter but can still be used optionally
* Optional curly brackets – if the expression body consists of a single statement, then the curly brackets can be removed
* Return keyword is optional. The compiler will return the value of the expression if it contains a single statement. If there are curly brackets around the expression body, then it will require a return statement whether it consists of a single statement or not
* Using a lambda expression first requires a functional interface that contains a method that it will override. There are many classes that are already built into Java that can be used as a functional interface, such as *Runnable*and *Comparator*:

@FunctionalInterface

public interface Comparator<T> {

int compare(T o1, T o2);

// contains other methods but compare is what needs to be overridden

}

* Before Java 8, there were two ways to use classes and override an interface’s behavior:
  + Make a new class and implement it and its method. Unless it is a piece of code that is going to be used a lot, this can be seen as overkill and could lead to lots of tiny classes that are only used once.
  + Use an Anonymous class and create an instance of the interface in the current code where it is needed and override the method there. For a piece of code that is used once, this makes more sense, as you get the functionality you need without the hassle and clutter of making a new class.
  + After the introduction of Java 8 lambda expressions, we can achieve the same with fewer lines of code
* Option 1: (Concrete class)

public class MyRunnable implements Runnable {

@Override

public void run() {

System.out.println("I have implemented Runnable");

}

public static void main(String args[]) {

MyRunnable runnable = new MyRunnable();

runnable.run();

}

}

* Option 2: (Anonymous class)

Runnable runnable = new Runnable() {

@Override

public void run() {

System.out.println("I have implemented Runnable");

}

};

runnable.run();

* Option 3: (lambda expression)

Runnable runnable = () -> System.out.println("I have implemented Runnable");

runnable.run();

* A lambda constitutes a comma separated list of formal parameters/or nothing enclosed in parenthesis as we would define in a method declaration, followed by an arrow token pointing to the code to execute.

**Some more features of Java 8**

* Default methods

Java8 allows method bodies to interfaces called default methods. These methods are implicitly added to every class which implements the interface. This enables you to add functionality to existing libraries without breaking code.