Java Functional Programming with Lambdas and Streams

Course Syllabus

- **Unit 1: Introduction to the Functional Programming**
  - Functional Programming with Java - Course Overview
  - Functional Programming with Java - Exploring GitHub Repo and Installations
- **Unit 2: Introduction to Functional Programming with Java**
  - Getting Started with Functional Programming with Java
  - Improving Java Functional Program with filter
  - Use Lambda Expression to enhance Functional Program
  - Using map in Functional Programs - with Exercises
  - Exercise on Functional Programming with Streams, Filters and Lambdas
- **Unit 3: Playing with Streams**
  - Stream Operations - Calculate Sum using reduce
  - Playing with reduce
  - Exploring Streams with Puzzles in JShell
  - Exercises on Functional Programming with Streams and reduce
  - Stream Operations - distinct and sorted
  - Using Comparators to Sort Streams with sorted
  - Collecting Stream Elements to List using collect
  - Reviewing Streams - Intermediate and Stream Operations
- **Unit 4: Java Functional Interfaces and Lambdas**
  - Getting Started with Functional Interfaces - Predicate, Consumer and Function
  - Do Exercises with Functional Interfaces - BinaryOperator
  - Doing Behaviour Parameterization with Functional Programming
  - Exercise with Behaviour Parameterization
  - Exploring Supplier and UnaryOperator Functional Interfaces
  - Exploring BiPredicate, BiFunction, BiConsumer, and Primitive Functional Interfaces
  - Playing Puzzles with Functional Interfaces and Lambdas
  - Exploring Method References with Java
- **Unit 5: Java Functional Programming with Custom Classes**
  - Creating Custom Class Course with some Test Data
  - Playing with allMatch, noneMatch and anyMatch
  - Sorting courses with sorted and creating Comparators
  - Playing with skip, limit, takeWhile and dropWhile
  - Finding top, max and min courses with max, min, findFirst and findAny
  - Playing with sum, average and count
  - Grouping courses into Map using groupingBy
- **Unit 6: Java Functional Programming**
  - Creating Streams using Stream of method and for Arrays
  - Creating Streams for First 100 Numbers, Squares of Numbers and More
  - Doing Big Number calculations with BigInteger
- **Unit 7: Playing further with Java Functional Programming**
  - Joining Strings with joining and Playing with flatMap
  - Creating Higher-Order Functions
  - FP and Performance - Intermediate Stream Operations are Lazy
  - Improving Performance with Parallelization of Streams
- **Unit 8: Functional Programming makes Java Easy**
  - Modifying lists with replaceAll and removeIf
  - Playing with Files using Functional Programming
  - Playing with Threads using Functional Programming
  - Using Functional Programming in Java Applications

Author – Abhinav (GitHub - @abhinavg916)

Unit 1 – Introduction to Functional Programming

- Functional Programming is a paradigm shift in coding which means it is focused on changing the way of problem solving.
- Use Java above 9 (Java 9+), but it has allowed to be used in Java 8 but JShell is not provided

Author – Abhinav (GitHub - @abhinavg916)

Unit 2 – Introduction to Functional Programming with Java

Task 1 – Print all the numbers from a list

forEach()

- It consumes the stream
- Traditional Approach focuses on How to solve a problem
- Functional Approach focuses on What to do for a problem

```
1.  public static void main (String[] args) {
2.            printAllNumbersInListStructure(List.of(12, 9, 13, 4, 6, 2, 4, 12, 15));
3.    }
4.
5.    private static void printAllNumbersInListStructure(List<Integer> numbers) {
6.            // To think - How to loop the numbers?
7.            for (int number: numbers) {
8.                    System.out.println(number);
9.            }
10.           // This means List -> Loop -> Operation
11.   }
```

Method References

- It is the simplified version of Lambda Expression for calling methods.
- To call operation for each element of forEach loop, we need to define what needs to be done or as behaviour. To define the behaviour, method references are used.
- Method References is widely used over static methods, but can be used for non-static methods as well
- Method References Syntax
    - For Static Methods

      ClassName::MethodName
    - For Non Static Methods

      new ClassName()::MethodName

```
1.  public static void main(String[] args) {
2.          // Approach - 2 : Functional Way
3.  printAllNumbersInListFunctional(List.of(12, 9, 13, 4, 6, 2, 4, 12, 15));
4.  }
5.  private static void print(int number) {
6.            System.out.println(number);
7.  }
8.  private static void printAllNumbersInListFunctional(List<Integer>numbers) {
9.            // To think - What to do?
10.           // Convert list of numbers into stream of numbers
11.           // [12, 9, 13, 4, 6, 2, 4, 12, 15]
12.           // That means it should come like in sequence
13.           // 12
14.           // 9
15.           // 13
16.           // 4
17.           // ... so on
18.
19.           numbers.stream().forEach(FP01Functional::println); // Use of Method References
20.           // This means List -> Streams -> Operation
21. }
```

- Improvised Functional Approach, no need of print method

```
1.  public static void main (String[] args) {
2.            printAllNumbersInListFunctional(List.of(12, 9, 13, 4, 6, 2, 4, 12, 15));
3.  }
4.  private static void printAllNumbersInListFunctional(List<Integer> numbers) {
5.            numbers.stream().forEach(System.out::println); // Use of Method References
6.  }
```

Author – Abhinav (GitHub - @abhinavg916)

Task 2 – Print even numbers from a list

filter()

- It contains the Predicate functions
- Traditional Way

```
1.  public static void main(String[] args) {
2.            List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
3.            printEvenNumbersInListStructure(numbers);
4.    }
5.  private static void printEvenNumbersInListStructure(List<Integer> numbers) {
6.            // To think - How to loop the numbers?
7.            for (int number : numbers) {
8.                    if (number % 2 == 0)
9.                            System.out.println(number);
10.           }
11.   }
```

- Functional Way

```
1.  public static void main(String[] args) {
2.            List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
3.            printEvenNumbersInListFunctional(numbers);
4.    }
5.  //        private static boolean isEven(int number) {
6.  //                return number % 2 == 0;
7.  //        }
8.  private static void printEvenNumbersInListFunctional(List<Integer> numbers) {
9.            // To think - What to do?
10. //        numbers.stream().filter(FP02Functional::isEven) // Use of Filter
11. //        .forEach(System.out::println); // Use of Method References
12.           // Improvised Way
13.           numbers.stream().filter(number -> number % 2 == 0) // Use of Lambda Expression
14.                           .forEach(System.out::println);
15.   }
```

Lambda Expression

- It is the simpler way of representation of method definitions
- For Multiple lines of coding in Lambda Expressions, {} are used.
- Syntax

  Parameter -> Operation
- Structed Way

```
1.  private static boolean isEven(int number) {
2.            return number % 2 == 0;
3.    }
```

- Lambda Expression

```
1.  number -> number % 2 == 0
```

Author – Abhinav (GitHub - @abhinavg916)

Task 3 - Print the squares of even numbers in the list

map()

- It maps the operation with the help of Lambda Expression on an element
- It contains the mapper function

```
1. public static void main(String[] args) {
2.           List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
3.           printSquaresofEvenNumbersInListFunctional(numbers);
4.   }
5. private static void printSquaresofEvenNumbersInListFunctional(List<Integer>
   numbers) {
6.           numbers.stream()
7.           .filter(number -> number % 2 == 0) // Use of Lambda Expression
8.           .map(x -> x*x) // Use of map for mapping x -> x*x
9.   .forEach(System.out::println);
10.  }
```

Author – Abhinav (GitHub - @abhinavg916)

Unit 3 – Playing with Streams

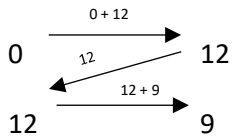
Task 4 - Find the sum of all the numbers in the list

reduce()

- It is used for combining and uses accumulator function
- Reduction operations parallelize more gracefully which means they don't need additional synchronization and with greatly reduced risk of data races.
- Parallelising enables to make efficient use of multicore processors
- It takes two arguments
    - First argument is called as Identity Value (Initial value)
    - Second argument is called as Accumulation Function
- For Example:

```
1. public static void main(String[] args) {
2.          List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
3.          printSumOfAllNumbersInListStructure(numbers);
4.   }
5. private static int sum(int a, int b) {
6.          return a + b;
7.   }
8.  private static void printSumOfAllNumbersInListStructure(List<Integer> numbers) {
9.          // Combine each one of them into one result i.e One value
10.         // 0 and (a,b) -> a + b
11.         // Stream of number -> One result value
12.         numbers.stream().reduce(0, FP04Functional::sum); // Use of Reduce
13.  }
```

- Working of reduce() with the output



12	9
21	13
34	4
38	6
44	2
46	4
50	12

- Improvised Way (Use of Lambda Expression)

```
1. private static int printSumOfAllNumbersInListStructure(List<Integer> numbers) {
2.          return numbers.stream().reduce(0, (a,b) -> a+b);
3. }
```

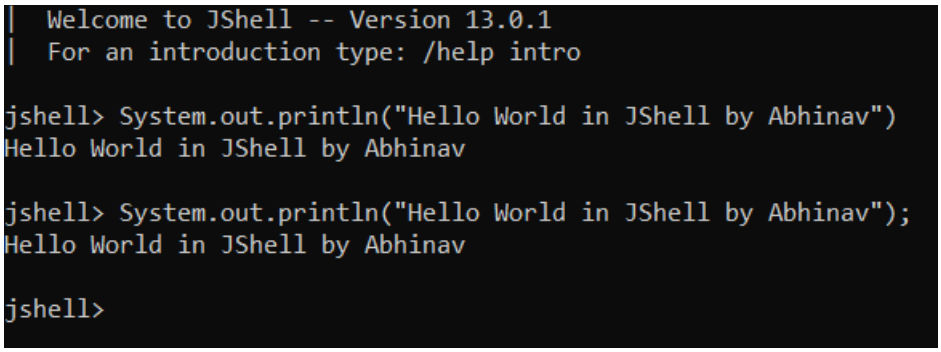
```
1. private static int printSumOfAllNumbersInListStructure(List<Integer> numbers) {
2.  return numbers.stream().reduce(0, Integer::sum);
3.   }
```

- Integer class has inbuilt sum static function which can also be used.

Author – Abhinav (GitHub - @abhinavg916)

JShell

- It was introduced in Java 9
- Before using JShell, make sure Java is installed and path is configured.
- It offers command line shell for java programming
- ; operator is not compulsory in JShell and can execute statement without it
- For Example:




```
1. jshell> List<Integer> numbers = List.of(12,9,13,4,6,2,4,12,15)
2. numbers ==> [12, 9, 13, 4, 6, 2, 4, 12, 15]
3.
4. jshell> numbers.stream().reduce(0, (a,b) -> a+b)
5. $4 ==> 77
6.
7. jshell> numbers.stream().reduce(0, (a,b) -> a)
8. $5 ==> 0
```

Task 5 - Find the maximum and minimum number in the list

```
jshell> List<Integer> numbers = List.of(12,9,13,4,6,2,4,12,15)
numbers ==> [12, 9, 13, 4, 6, 2, 4, 12, 15]

jshell> numbers.stream().reduce(0, (a,b) -> a>b?a:b)
$6 ==> 15
```

- But it doesn't work well in all scenarios because during finding minimum number, minimum number can be negative and we get answer 0, which is wrong.
- reduce(0, (a,b) -> a>b?a:b); // Here, 0 is Identity Value (Initial Value) which means always compared with 0
- To overcome this, Integer.MIN_VALUE and Integer.MAX_VALUE is used

```
jshell> numbers.stream().reduce(Integer.MIN_VALUE, (a,b) -> a>b?a:b)
$8 ==> 15

jshell> numbers.stream().reduce(Integer.MIN_VALUE, (a,b) -> a<b?a:b)
$8 ==> -2147483648

jshell> numbers.stream().reduce(Integer.MAX_VALUE, (a,b) -> a>b?b:a)
$10 ==> 2

jshell> numbers
numbers ==> [12, 9, 13, 4, 6, 2, 4, 12, 15]
```

- Wrong answer because of comparison with identity value (initial value).
- For finding minimum element, it should be compared with the maximum value
- For finding maximum element, it should be compared with the minimum element.

Author – Abhinav (GitHub - @abhinavg916)

Task 6 - To find the distinct number in the list

distinct()

- Method to find the distinct numbers

```
jshell> numbers
numbers ==> [12, 9, 13, 4, 6, 2, 4, 12, 15]

jshell> numbers.stream().distinct().forEach(System.out::println)
12
9
13
4
6
2
15
```

Author – Abhinav (GitHub - @abhinavg916)

Task 7 - To sort the numbers in the list

sorted()

- It sorts the list in ascending order
- It works with both numbers and strings

```
jshell> numbers
numbers ==> [12, 9, 13, 4, 6, 2, 4, 12, 15]

jshell> numbers.stream().sorted().forEach(System.out::println)
2
4
4
6
9
12
12
13
15
```

- To remove the duplicate elements, distinct() is used before sorted()

```
jshell> numbers.stream().distinct().sorted().forEach(System.out::println)
2
4
6
9
12
13
15
```

- For sorting of strings

```
jshell> List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices",
   "AWS", "PCF", "Azure", "Docker", "Kubernetes");
courses ==> [Spring, Spring Boot, API, Microservices, AWS, PCF, Azure, Docker, Kubernetes]

jshell> courses.stream().sorted().forEach(System.out::println)
API
AWS
Azure
Docker
Kubernetes
Microservices
PCF
Spring
Spring Boot
```

Comparator

- To use the custom algorithm for sorting the elements, Comparator is used and type must be primitive values
- natualOrder() method sorts in ascending
- reverseOrder() method sorts in descending

```
jshell>courses.stream().sorted(Comparator.naturalOrder()).forEach(System.out::println)
API
AWS
Azure
Docker
Kubernetes
Microservices
PCF
Spring
Spring Boot

jshell> courses.stream().sorted(Comparator.reverseOrder()).forEach(System.out::println)
Spring Boot
```

```
14. Spring
15. PCF
16. Microservices
17. Kubernetes
18. Docker
19. Azure
20. AWS
21. API
```

- To define own logic, comparing() is used and lambda expression is put in it.
- For Example, sorting elements on the basis of string length

```
1.  jshell> courses.stream().sorted(Comparator.comparing(str ->
    str.length())).forEach(System.out::println)
2.  API
3.  AWS
4.  PCF
5.  Azure
6.  Spring
7.  Docker
8.  Kubernetes
9.  Spring Boot
10. Microservices
```

Task 8 – Create list from another list

collect()

- It collects the elements from stream and put them to another list
- It uses Collectors functions
- It uses Collectors class present in java.util package

```
1.  public static void main(String[] args) {
2.            List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
3.            List<Integer> squaredNumbers = squaredList(numbers);
4.            System.out.println(squaredNumbers);
5.    }
6.  private static List<Integer> squaredList(List<Integer> numbers) {
7.            return numbers.stream()
8.                    .map(x -> x*x)
9.                    .collect(Collectors.toList());
10.   }
```

Stream Operation

- Operations on streams is categorized in two types:
    - Intermediate – It executes on a stream and returns another stream (stream of the modified values) through it i.e. Methods that returns Stream <T>
        - For Example: distinct(), sorted(), map(), filter(), etc.
    - Terminal – It consumes the stream and returns nothing i.e. Methods that returns void, R, or other than Stream <T>
        - For Example: void(), collect(), reduce(), etc.

Author – Abhinav (GitHub - @abhinavg916)

Unit 4: Java Functional Interfaces and Lambdas

Behind the scenes working of Lambda Expressions

- **Scenario I:** filter(), map(), forEach() take logic as input in form of Lambda Expressions

```
1. List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
2. numbers.stream().filter(x -> x%2==0).map(x -> x*x).forEach(System.out::println);
```

- **Scenario II:** filter(), map(), forEach() take method as input without the use of Lambda Expressions

```
1. numbers.stream().filter(FP11::isEven).map(FP11::map).forEach(FP11::print);
```

```
1. /*
2.              - Equivalent method for filter()
3.              public static boolean isEven(int x){
4.                      return x % 2 == 0;
5.              }
6.
7.              - Equivalent method for map()
8.              public static int squared (int x){
9.                      return x*x;
10.             }
11.
12.             - Equivalent method for forEach()
13.             public static void print(int x){
14.             System.out.println(x);    OR      forEach( x -> System.out.println(x));
15.    */
```

- Method references are used to simply the lambda expression for method calling.
- **Scenario III:** filter(), map(), forEach() in form of Functional Interfaces
- There are classes behind the lambda expressions like Consumer, Predicate, Function and instance of these classes were created when used them in this way:

```
1. // Extract as local variable for filter(), map(), forEach()
2.   Predicate<Integer> isEvenPredicate = x -> x % 2 == 0; // Predicate Function
3.   Function<Integer, Integer> squaredMapper = x -> x * x; // Function Function
4. Function<Integer, String> stringOutputMapper = x -> x + "*"; // Function Function which
   returns string as output
5.   Consumer<Integer> printAction = System.out::println;
```

```
1. Predicate<Integer> isEvenPredicate2 = new Predicate<Integer>() {
2.              @Override
3.              public boolean test(Integer x) {
4.                      return x%2==0;
5.              }
6.    };
```

```
1. Function<Integer, Integer> squaredMapper2 = new Function<Integer, Integer>(){
2.              @Override
3.              public Integer apply(Integer x) {
4.                      return x*x;
5.              }
6.    };
```

```
1. Consumer<Integer> printAction2 = new Consumer<Integer>() {
2.              @Override
3.              public void accept(Integer x) {
4.                      System.out.println(x);
5.              }
6.    };
```

```
1.   numbers.stream().filter(isEvenPredicate2).map(squaredMapper2).forEach(printAction2);
```

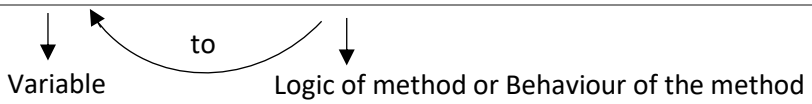
- Predicate, Consumer, Function, Binary Operator, BiFunction are all Functional Interfaces

Functional Interface

- An interface that has only one abstract method.
- For Example: Consumer, Predicate, Function, Binary Operator, BiFunction, Supplier, Unary Operator, etc.
- All the functional interfaces are present in java.util.function package.
- From above code snippet, we can understand the isEvenPredicate2 return value is being sent to the filter()
- Java as a programming language focuses much more backward compatibility and thus functional programming was introduced late due to not to break earlier existing features of Java.
- So, Lambda Expression is treated like an object in Functional Programming because objects are first class citizen of a class in Java and functions are first class citizens in Python and JavaScript rather than object.
- Python and JavaScript were built in start to treat functions as first-class citizens.
- Just like you store data into variable, just like you pass data to a method, you can pass functions to a method, you can actually store a function into a variable, and things like that in Python.
- **Predicate** – It represents a predicate (boolean-valued function) of one argument
- It is typically (mostly) used to test something and is used in filter()
- **Function** – It represents a function that accepts one argument and produces a result.
- It is used to when one input is taken and corresponding to it one output is returned. It is used in map()
- **Consumer** – It represents an operation that accepts a single input argument and returns no result back.
- It is used when input has to be only consumed. It is used in forEach()
- **Binary Operator** – It represents an operation upon two operands of the same type, produces a result of the same type as the operands. It is used in reduce()
- The abstract function inside the functional interface is called as Function Descriptor.
- For Example: In Predicate is test(T t), In Function is apply(T t), In Consumer is accept(T t), Binary Operator extends BiFunction which has apply(T t, U u).

Behaviour Parameterization

```
1. Predicate<Integer> isEvenPredicate = x -> x % 2 == 0;
```



- Traditionally: Behaviour of Method -> Variable -> Variable is sent as an Argument to other method. This means Data is sent as a parameter into the method.
- Passing the logic of the method as an argument to another method i.e. Behaviour is sent as an argument and is called as Behaviour Parameterisation.

```
1. public static void main(String[] args) {
2.             List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
3.             // Redundancy in code
4.             numbers.stream().filter(x -> x % 2 == 0).forEach(System.out::println);
5.             numbers.stream().filter(x -> x % 2 != 0).forEach(System.out::println);
6.
7.             // Solution
8.             Predicate<? super Integer> evenPredicate = x -> x % 2 == 0;
9.             filterandPrint(numbers, evenPredicate);
10.            Predicate<? super Integer> oddPredicate = x -> x % 2 != 0;
11.            filterandPrint(numbers, oddPredicate);
12.
13.            // Inline Method
14.            filterandPrint(numbers, x -> x % 2 == 0); // Even Numbers
15.            filterandPrint(numbers, x -> x % 2 != 0); // Odd Numbers
16.            filterandPrint(numbers, x -> x % 3 == 0); // Multiple of 3
17.    }
18. private static void filterandPrint(List<Integer> numbers, Predicate<? super Integer>
    predicate) {
19.            numbers.stream().filter(predicate).forEach(System.out::println);
20.  }
```

Author – Abhinav (GitHub - @abhinavg916)

- This helps in reducing redundant code and using one method to get the multiple answers based on the logic sent as parameter.
- This has enabled function as first class citizens in Java. Now, pass methods into another methods, store methods into a variable, functions can be sent as argument, return method from another method.
- Steps:
    - Extract the logic to local variable using Refracting
    - Extract the functional programming statement to Method using Refracting
    - Inline the function for ease
    - Implement different logics for behaviour parameterisation

Supplier and Unary Operator Functional Interfaces

- **Supplier** – It takes no input and returns one output
- It contains abstract function as get()

```
1.  Supplier<Integer> randomIntegerSupplier = () -> 2; // () represents no parameters and
    return 2 everytime
2.              Supplier<Integer> randomIntegerSupplier2 = () -> {
3.                      Random random = new Random();
4.                      return random.nextInt(1000);
5.
6.              }; // For Multiple lines of coding in Lambda Expressions, {} are used.
7.              System.out.println(randomIntegerSupplier.get());
8.              System.out.println(randomIntegerSupplier2.get());
```

- **Unary Operator** – It represents an operation on a single operand that produces a result of the same type.
- It contains abstract function as identity () and apply(T t)

```
1.  UnaryOperator<Integer> unaryOperator = (x) -> x * 3;
2.  System.out.println(unaryOperator.apply(10)); // 10 as input and 30 as output
```

BiPredicate, BiFunction, BiConsumer and Primitive Functional

- **BiPredicate** - It is similar to Predicate but it takes two inputs and returns output of type Boolean
- It contains abstract function as test (T t)

```
1.  BiPredicate<Integer, String> biPredicate = (number, str) -> true;
2.              BiPredicate<Integer, String> biPredicate2 = (number, str) -> { // Multiple line
    lambda expression
3.                      return number < 10 && str.length() > 5;
4.              };
5.
6.              System.out.println(biPredicate.test(5, "in28Minutes"));
7.              System.out.println(biPredicate2.test(10, "in28Minutes"));
```

- **BiFunction –** It is similar to Function but it takes two inputs and return one output.
- Return type is mandatory to be stated which can String, Boolean, etc depending upon the logic, otherwise compile time error.
- It contains abstract function as apply()

```
1.    BiFunction<Integer, String, String> biFunction = (number, str) -> {
2.                      return number + " " + str;
3.              };
4.
5.              System.out.println(biFunction.apply(5, "in28Minutes"));
```

- **BiConsumer –** It is similar to Consumer but takes two inputs, consumes it and does not return anything.
- It contains abstract function as accept()

```
1. BiConsumer<Integer, String> biConsumer = (s1, s2) -> {
2.                     System.out.println(s1);
3.                     System.out.println(s2);
4.             };
5.
6. // System.out.println(biConsumer.accept(15, "in28Minutes")); // Error because it returns
   nothing and sysout cannot be used.
7.             biConsumer.accept(15, "in28Minutes");
```

- We have the following more functional interfaces, so as to represent the operations using the primitives.
- They are as follows:
    - IntBinaryOperator
    - IntConsumer
    - IntFunction
    - IntPredicate
    - IntSupplier
    - InttoDoubleFunction
    - InttoLongFunction
    - IntUnaryOperator
- For Example:

```
1. BinaryOperator<Integer> sumBinaryOperator = (x, y) -> x + y; // Uses Integer (Wrapper)
   classes
2. IntBinaryOperator intBinaryOperator = (x, y) -> x + y; // Use int data type, No Boxing
```

- With the use of Wrapper classes, Boxing and Unboxing comes into picture and is inefficient.
- It is better to use primitive lambda expressions.
- Until now, objects of wrapper classes were being used.
- When using Primitive Functional Interfaces, make sure to use primitive operations.

Puzzles and Questions

```
1. //jshell> Consumer<String> consumer = () -> {}
2. //|  Error:
3. //|  incompatible types: incompatible parameter types in lambda expression
4. //|  Consumer<String> consumer = () -> {};  // REASON: Because it expects one input
5. //|                               ^------^
6. //
7. //jshell> Consumer<String> consumer = (str) -> {}
8. //consumer ==> $Lambda$17/0x0000000800bb2c40@3c0ecd4b
9. //
10. //jshell> Consumer<String> consumer = (str) -> System.out::println
11. //|  Error:
12. //|  incompatible types: bad return type in lambda expression
13. //|      void is not a functional interface
14. //|  Consumer<String> consumer = (str) -> System.out::println;
15. //|                                       ^-----------------^
16. //
17. //jshell> Consumer<String> consumer = System.out::println
18. //
19. //jshell> Consumer<String> consumer = (str) -> System.out.println(str)
20. //consumer ==> $Lambda$18/0x0000000800bb3840@5f341870
21. //
22. //jshell> BiConsumer<String, String> consumer = (str, str2) -> System.out.println(str)
23. //consumer ==> $Lambda$19/0x0000000800bb3c40@271053e1
24. //
25. //jshell> Supplier<String> supplier = () -> "Abhinav"
26. //supplier ==> $Lambda$20/0x0000000800bb4840@6500df86
27. //
28. //jshell> Supplier<String> supplier = () -> {"Abhinav"}
29. //|  Error:
30. //|  not a statement
31. //|  Supplier<String> supplier = () -> {"Abhinav"};          // REASON: Wrong Syntax
32. //|                                     ^-------^
```

```
33. //|  Error:
34. //|  ';' expected
35. //|  Supplier<String> supplier = () -> {"Abhinav";};         // REASON: Wrong Syntax
36. //|                                                  ^
37. //
38. //jshell> Supplier<String> supplier = () -> {return "Abhinav"};
39. //|  Error:
40. //|  ';' expected
41. //|  Supplier<String> supplier = () -> {return "Abhinav"};    // REASON : Wrong Syntax
42. //|                                                        ^
43. //
44. //jshell> Supplier<String> supplier = () -> {return "Abhinav";}
45. //supplier ==> $Lambda$20/0x0000000800bb4840@6500df86
46. //
47. //jshell> Consumer<String> consumer = (str) -> System.out.println(str);
    System.out.println(str);
48. //consumer ==> $Lambda$21/0x0000000800bb4c40@4cf777e8
49. //|  Error:
50. //|  cannot find symbol
51. //|    symbol:   variable str
52. //|   System.out.println(str); // REASON: Wrong syntax of writing multiline lambda exprsn.
53. //|                       ^-^
54. //
55. //jshell> Consumer<String> consumer = (str) -> {System.out.println(str);
    System.out.println(str);}
56. //consumer ==> $Lambda$22/0x0000000800bb5840@3fee9989
```

- It is necessary to define the type of all the parameter in the lambda expression if once defined for a parameter.

```
1. List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
2.
3. Predicate<Integer> isEvenPredicate = (Integer x) -> x % 2 == 0; // Type defined for the
   parameter in lambda expression
4.
5. BinaryOperator<Integer> binaryOperator = (Integer x, Integer y) -> x + y; // Type defined
   for all the parameter in lambda expression
6.
7. BinaryOperator<Integer> binaryOperator2 = (x, y) -> x + y; // Type Inference, Compiler
   automatically infer type as 'Integer' defined in Binary Operator definition
8.
9. BinaryOperator<Integer> binaryOperator3 = (String x, String y) -> x + y; // Compile Error
   because of wrong type
```

Method References

- It is primarily used for simplifying lambda expression over methods.
- It can even be used over to call things on a specific object.
- It can also be used for constructor references.
- Syntax for static methods:

  ClassName::MethodName

```
1. private static void print(String str) {
2.           System.out.println(str);
3.   }
4.
5.   public static void main(String[] args) {
6.   List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices", "AWS",
   "PCF", "Azure", "Docker", "Kubernetes");
7.   courses.stream()
8.   .map(str -> str.toUpperCase())
9.   .forEach(str -> System.out.println(str));
10.
11.  courses.stream()
12.  .map(str -> str.toUpperCase())
13.  .forEach(System.out::println); // Use of Method References, println() is called each
   time for each element in the stream
14.
15.  courses.stream()
```

```
16.   .map(str -> str.toUpperCase())
17.   .forEach(FP16MethodReferences::print); // Use of Method References and print is user
    defined static method
18.
19.   courses.stream()
20. //.map(str -> str.toUpperCase())
21.   .map(String::toUpperCase) // Use of Method References for calling over the object and
    toUpperCase() is not a static method in String class
22.   .forEach(FP16MethodReferences::print);  // print is UDF static method
23.
24.   Supplier<String> supplier = () -> new String(); // Lambda Expression which creates the
    string
25.   Supplier<String> supplier2 = String::new; // Constructor References
26.           }
```

Unit 5: Java Functional Programming with Custom Classes

Custom Class

```
1. class Courses {
2.   private String name;
3.   private String category;
4.   private int reviewScore;
5.   private int noOfStudents;
6.
7.   public Courses(String name, String category, int reviewScore, int noOfStudents) {
8.           super();
9.           this.name = name;
10.          this.category = category;
11.          this.reviewScore = reviewScore;
12.          this.noOfStudents = noOfStudents;
13.  }
14.  public String getName() {
15.          return name;
16.  }
17.  public void setName(String name) {
18.          this.name = name;
19.  }
20.  public String getCategory() {
21.          return category;
22.  }
23.  public void setCategory(String category) {
24.          this.category = category;
25.  }
26.  public int getReviewScore() {
27.          return reviewScore;
28.  }
29.  public void setReviewScore(int reviewScore) {
30.          this.reviewScore = reviewScore;
31.  }
32.  public int getNoOfStudents() {
33.          return noOfStudents;
34.  }
35.  public void setNoOfStudents(int noOfStudents) {
36.          this.noOfStudents = noOfStudents;
37.  }
38.  public String toString() {
39.          return name + ":" + noOfStudents + ":" + reviewScore;
40.  }
41. }
```

```
1.   List<Courses> courses = List.of(
2.                              new Courses("Spring", "Framework", 98, 20000),
3.                              new Courses("Spring Boot", "Framework", 95, 18000),
4.                              new Courses("API", "Microservices", 97, 22000),
5.                              new Courses("Microservices", "Microservices", 96, 25000),
6.                              new Courses("FullStack", "FullStack", 91, 14000),
7.                              new Courses("AWS", "Cloud", 92, 21000),
8.                              new Courses("Azure", "Cloud", 99, 21000),
9.                              new Courses("Docker", "Cloud", 92, 20000),
10.                             new Courses("Kubernates", "Cloud", 91, 2000)
11.          );
```

- More Stream Operations for comparing:
    - **allMatch()** – It return true if every element matches otherwise false. It uses Predicate.
    - **noneMatch()** – It returns true if none of the element matches otherwise false. It uses Predicate.
    - **anyMatch()** – It returns true if any one of the element matches otherwise false. It uses Predicate.

Task 9 - Courses having good reviews

```
1.    Predicate<Courses> reviewScoreGreaterThan95Predicate = course -> course.getReviewScore() > 95;
2.    Predicate<Courses> reviewScoreGreaterThan90Predicate = course -> course.getReviewScore() > 90;
3.    Predicate<Courses> reviewScoreLessThan90Predicate = course -> course.getReviewScore() < 90;
4.
5.    //allMatch
6.    System.out.println(courses.stream().allMatch(course -> course.getReviewScore() > 90)); // true
7.    System.out.println(courses.stream().allMatch(course -> course.getReviewScore() > 95)); // false
    beacuse not all are above 95
8.    System.out.println(courses.stream().allMatch(reviewScoreGreaterThan95Predicate)); // false
9.
10.   //noneMatch
11.   System.out.println(courses.stream().noneMatch(course -> course.getReviewScore() > 95)); // false
    because none  of the courses must be greater than 95
12.   System.out.println(courses.stream().noneMatch(reviewScoreLessThan90Predicate)); // true because
    none of the courses is less than 90
13.
14.   //anyMatch
15.   System.out.println(courses.stream().anyMatch(reviewScoreGreaterThan95Predicate)); // true
16.   System.out.println(courses.stream().anyMatch(reviewScoreLessThan90Predicate)); // false
```

Author – Abhinav (GitHub - @abhinavg916)

Task 10 – Sorting Custom Classes (Courses)

```
1.  class Course {
2.    private String name;
3.    private String category;
4.    private int reviewScore;
5.    private int noOfStudents;
6.
7.    public Course(String name, String category, int reviewScore, int noOfStudents) {
8.            super();
9.            this.name = name;
10.           this.category = category;
11.           this.reviewScore = reviewScore;
12.           this.noOfStudents = noOfStudents;
13.   }
14.   public String getName() {
15.           return name;
16.   }
17.   public void setName(String name) {
18.           this.name = name;
19.   }
20.   public String getCategory() {
21.           return category;
22.   }
23.   public void setCategory(String category) {
24.           this.category = category;
25.   }
26.   public int getReviewScore() {
27.           return reviewScore;
28.   }
29.   public void setReviewScore(int reviewScore) {
30.           this.reviewScore = reviewScore;
31.   }
32.   public int getNoOfStudents() {
33.           return noOfStudents;
34.   }
35.   public void setNoOfStudents(int noOfStudents) {
36.           this.noOfStudents = noOfStudents;
37.   }
38.   public String toString() {
39.           return name + ":" + noOfStudents + ":" + reviewScore;
40.   }
41. }
```

```
1.  List<Course> courses = List.of(
2.                              new Course("Spring", "Framework", 98, 20000),
3.                              new Course("Spring Boot", "Framework", 95, 18000),
4.                              new Course("API", "Microservices", 97, 22000),
5.                              new Course("Microservices", "Microservices", 96, 25000),
6.                              new Course("FullStack", "FullStack", 91, 14000),
7.                              new Course("AWS", "Cloud", 92, 21000),
8.                              new Course("Azure", "Cloud", 99, 21000),
9.                              new Course("Docker", "Cloud", 92, 20000),
10.                             new Course("Kubernates", "Cloud", 91, 2000)
11.           );
```

- Comparator are used.
- Make sure to use primitive values with Comparator.

```
1.  Comparator<Course> comparingByNumberOfStudentsIncreasing =
    Comparator.comparing(Course::getNoOfStudents);
2.  System.out.println(courses.stream().sorted(comparingByNumberOfStudentsIncreasing).collect
3.  (Collectors.toList()));
```

```
1.  Comparator<Course>          comparingByNumberOfStudentsDecreasing =
    Comparator.comparing(Course::getNoOfStudents).reversed();
2.  System.out.println(courses.stream().sorted(comparingByNumberOfStudentsDecreasing).collect(
    Collectors.toList()));
```

```
3. // [Microservices:25000:96, API:22000:97, AWS:21000:92, Azure:21000:99, Spring:20000:98,
   Docker:20000:92, Spring Boot:18000:95, FullStack:14000:91, Kubernates:2000:91]
```

- If the courses have equal number of students then sorting according to the review score.

```
1. Comparator<Course>comparingByNumberOfStudentsAndReviewScore = Comparator
2.                     .comparing(Course::getNoOfStudents)
3.                     .thenComparing(Course::getReviewScore)
4.                     .reversed();
5. System.out.println(courses.stream().sorted(comparingByNumberOfStudentsAndReviewScore)
6. .collect(Collectors.toList()));
7. // [Microservices:25000:96, API:22000:97, Azure:21000:99, AWS:21000:92, Spring:20000:98,
   Docker:20000:92, Spring Boot:18000:95, FullStack:14000:91, Kubernates:2000:91]
```

.comparing() vs .comparingInt()

- .comparingInt() is better because of primitive values where as .comparing() needs to check for boxing each time

```
1. Comparator<Course> comparingByNumberOfStudentsIncreasing =
   Comparator.comparingInt(Course::getNoOfStudents);
2. System.out.println(courses.stream().sorted(comparingByNumberOfStudentsIncreasing).collect(
   Collectors.toList()));
3.
4. Comparator<Course> comparingByNumberOfStudentsDecreasing =
   Comparator.comparingInt(Course::getNoOfStudents).reversed();
5. System.out.println(courses.stream().sorted(comparingByNumberOfStudentsDecreasing).collect(
   Collectors.toList()));
6.
7. Comparator<Course> comparingByNumberOfStudentsAndReviewScore = Comparator
8.                     .comparingInt(Course::getNoOfStudents)
9.                     .thenComparing(Course::getReviewScore)
10.                    .reversed();
11. System.out.println(courses.stream().sorted(comparingByNumberOfStudentsAndReviewScore).coll
   ect(Collectors.toList()));
```

Task 12 – Utility Operations

skip() – It skips the top required number of outputs

```
1. System.out.println(courses.stream()
2.                            .sorted(comparingByNumberOfStudentsAndReviewScore)
3.                            .skip(3)          // Top 3 are skipped
4.                            .collect(Collectors.toList()));
5. // [AWS:21000:92, Spring:20000:98, Docker:20000:92, Spring Boot:18000:95,
   FullStack:14000:91, Kubernates:2000:91]
```

limit() – It shows only the top required number of outputs

```
1. System.out.println(courses.stream()
2.                            .sorted(comparingByNumberOfStudentsAndReviewScore)
3.                            .limit(5)                 // Top 5 only
4.                            .collect(Collectors.toList()));
5. //[Microservices:25000:96, API:22000:97, Azure:21000:99, AWS:21000:92, Spring:20000:98]
```

```
1. System.out.println(courses.stream()
2.           .sorted(comparingByNumberOfStudentsAndReviewScore)
3.           .skip(3)          // Top 3 are skipped i.e Microservices, API and Azure
4.           .limit(5)         // Top 5 are shown
5.           .collect(Collectors.toList()));
6. //[AWS:21000:92, Spring:20000:98, Docker:20000:92, Spring Boot:18000:95,
   FullStack:14000:91]
```

takeWhile() – It takes all elements until an element which is to be target is found i.e as soon as it finds the target element, it stops and display the results.

- While condition is satisfied it keeps on executing and when condition is not satisfied it stops.
- It uses Predicate.

```
1. System.out.println(courses);
2. //[Spring:20000:98, Spring Boot:18000:95, API:22000:97, Microservices:25000:96,
   FullStack:14000:91, AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernates:2000:91]
3. System.out.println(courses.stream().takeWhile(course ->
   course.getReviewScore()>=95).collect(Collectors.toList()));
4. //[Spring:20000:98, Spring Boot:18000:95, API:22000:97, Microservices:25000:96]
```

dropWhile() – It skips the elements till the condition is satisfied and prints for when the condition is not satisfied.

- It is opposite of takeWhile()
- It uses Predicate.

```
1. System.out.println(courses);
2. //[Spring:20000:98, Spring Boot:18000:95, API:22000:97, Microservices:25000:96,
   FullStack:14000:91, AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernates:2000:91]
3. System.out.println(courses.stream().dropWhile(course ->
   course.getReviewScore()>=95).collect(Collectors.toList()));
4. //[FullStack:14000:91, AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernates:2000:91]
```

top() -

min() – It returns the first element in the list

```
1. System.out.println(courses);
2. System.out.println(courses.stream().min(comparingByNumberOfStudentsAndReviewScore));
3. //Optional[Microservices:25000:96]
```

```
1. System.out.println(courses);
2. System.out.println(courses.stream()
3.           .filter(reviewScoreLessThan90Predicate)
4.           .min(comparingByNumberOfStudentsAndReviewScore));
```

Author – Abhinav (GitHub - @abhinavg916)

```
5. //        Optional.empty
```

- Typically in Java, we are used to handle Null values on return. Using null values, we represent absence in our databases but it results in NullPointerExceptions
- **Optional** is a way to resolve NullPointerExceptions and letting the not to end abruptly. It returns Optional.empty
- Optional provides a way to check whether the result exists or not

```
1. System.out.println(courses);
2.            System.out.println(courses.stream()
3.            .filter(reviewScoreLessThan90Predicate)
4.            .min(comparingByNumberOfStudentsAndReviewScore)
5.            .orElse(new Course2 ("Kubernates", "Cloud", 91, 20000)));
6. //         Optional.empty
7. //         Kubernates:20000:91
```

max() – It returns the last element in the list

```
1. System.out.println(courses);
2. //[Spring:20000:98, Spring Boot:18000:95, API:22000:97, Microservices:25000:96,
   FullStack:14000:91, AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernates:2000:91]
3. System.out.println(courses.stream().max(comparingByNumberOfStudentsAndReviewScore));
4. //Optional[Kubernates:2000:91]
```

findFirst() -To find the first element that meets the criteria

- It returns Optional value

```
1.   System.out.println(courses.stream()
2.            .filter(reviewScoreGreaterThan95Predicate)
3.            .findFirst()
4.            );
5. //         Optional[Spring:20000:98]
```

findAny() – It finds any of the element from the stream that meets the criteria

- It is non-deterministic in nature of output
- It returns Optional value

```
1.  System.out.println(courses.stream()
2.   .filter(reviewScoreGreaterThan95Predicate)
3.   .findAny()
4.   );
5. //Optional[Spring:20000:98]
```

sum() – It calculates the sum

- It returns numeric value

```
1. System.out.println(courses.stream()
2.   .filter(reviewScoreGreaterThan95Predicate)
3.   .mapToInt(Course2::getNoOfStudents) // When sure about primitive return value then use
   primitive methods for avoiding boxing
4.   .sum() // Finds sum of total no. of students who are in courses of review score > 95
5.   );
6. //88000
```

average() – It calculates the average

- It returns Optional value

```
1. System.out.println(courses.stream()
2.            .filter(reviewScoreGreaterThan95Predicate)
3.            .mapToInt(Course2::getNoOfStudents)
```

```
4.              .average()
5.              );
6.  //          OptionalDouble[22000.0]
```

count() – It calculates the count

- It returns numeric value

```
1.  System.out.println(courses.stream()
2.              .filter(reviewScoreGreaterThan95Predicate)
3.              .mapToInt(Course2::getNoOfStudents)
4.              .count()
5.              );
6.  //4
```

max()

```
1.  System.out.println(courses.stream()
2.              .filter(reviewScoreGreaterThan95Predicate)
3.              .mapToInt(Course2::getNoOfStudents)
4.              .max()
5.              );
6.  //OptionalInt[25000]
```

min()

```
1.  System.out.println(courses.stream()
2.              .filter(reviewScoreGreaterThan95Predicate)
3.              .mapToInt(Course2::getNoOfStudents)
4.              .min()
5.              );
6.  //          OptionalInt[20000]
```

Task 13 – Grouping

groupingBy() – To categorize/group the similar element

```
System.out.println(courses.stream()
          .collect(Collectors.groupingBy(Course3::getCategory)));
/*{Cloud=[AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernates:2000:91],
   FullStack=[FullStack:14000:91], Microservices=[API:22000:97, Microservices:25000:96],
   Framework=[Spring:20000:98, Spring Bo     ot:18000:95]} */
```

counting()

- It maintains the key-value pair
- It internally use Hashmap

```
System.out.println(courses.stream()
  .collect(Collectors.groupingBy(Course3::getCategory, Collectors.counting())));
//{Cloud=4, FullStack=1, Microservices=2, Framework=2}
```

maxBy()

- It used to find the maximum element based on the condition
- It uses Comparator
- The below code snippet finds the maximum from each category

```
System.out.println(courses.stream()
          .collect(Collectors.groupingBy(Course3::getCategory,
Collectors.maxBy(Comparator.comparing(Course3::getReviewScore)))));
/*{Cloud=Optional[Azure:21000:99], FullStack=Optional[FullStack:14000:91],
  Microservices=Optional[API:22000:97], Framework=Optional[Spring:20000:98]} */
```

mapping()

- It uses to map
- The below code snippet maps the name of courses under each category

```
System.out.println(courses.stream()
          .collect(Collectors.groupingBy(Course3::getCategory,
          Collectors.mapping(Course3::getName, Collectors.toList() ))));
/*{Cloud=[AWS, Azure, Docker, Kubernates], FullStack=[FullStack],
  Microservices=[API, Microservices], Framework=[Spring, Spring Boot]} */
```

Author – Abhinav (GitHub - @abhinavg916)

Unit 6: Java Functional Programming

Creating Streams using stream of method and for Arrays

```
1. jshell> List<Integer> numbers = List.of(12,9,13,4,6,2,4,12,15);
2. numbers ==> [12, 9, 13, 4, 6, 2, 4, 12, 15]
3.
4. jshell> numbers.stream()
5. $2 ==> java.util.stream.ReferencePipeline$Head@14bf9759
6.
7. jshell> Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
8. $3 ==> java.util.stream.ReferencePipeline$Head@553f17c
9.
10. jshell> Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15).count()
11. $4 ==> 9
12.
13. jshell> Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15).reduce(0, Integer::sum)
14. $5 ==> 77
```

- List of numbers are converted to stream and then operations are applied on it. This means elements are boxed.

Stream.of()

- It creates the stream of numbers directly and operations are applied on it but it still uses boxing
- Reference type streams are used when Wrapper classes or custom classes are used

Primitive Streams

- Primitive Streams are faster in performance
- int or primitive type streams are used when primitive types are used where boxing does not occur
- It supports more methods natively for operations.

```
1. jshell> int[] numberArray = {12,9,13,4,6,2,4,12,15}
2. numberArray ==> int[9] { 12, 9, 13, 4, 6, 2, 4, 12, 15 }
3.
4. jshell> Arrays.stream(numberArray)
5. $2 ==> java.util.stream.IntPipeline$Head@4f7d0008
6.
7. jshell> Arrays.stream(numberArray).sum()
8. $3 ==> 77
9.
10. jshell> Arrays.stream(numberArray).average()
11. $4 ==> OptionalDouble[8.555555555555555]
12.
13. jshell> Arrays.stream(numberArray).min()
14. $5 ==> OptionalInt[2]
15.
16. jshell> Arrays.stream(numberArray).max()
17. $6 ==> OptionalInt[15]
```

```
1. jshell> Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15).average() // Not supported for reference
   type stream
2. |  Error:
3. |  cannot find symbol
4. |    symbol:   method average()
5. |  Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15).average()
6. |  ^-----------------------------------------^
```

- More examples of creation of primitive streams
  - range(a, n) – It generates numbers from a to n-1
  - rangeClosed(a, n) – It generates numbers from a to n
  - iterate() – It is used to customize the sequence of numbers and can lead to infinite. So, limit() is used to avoid the infinite.

Author – Abhinav (GitHub - @abhinavg916)

```
jshell> IntStream.range(1,10).sum()
$1 ==> 45

jshell> IntStream.rangeClosed(1,10).sum()
$2 ==> 55
jshell> IntStream.iterate(1, e -> e+2).limit(10).peek(System.out::println).sum()
1
3
5
7
9
11
13
15
17
19
$3 ==> 100
```

```
jshell> IntStream.iterate(2, e -> e*2).limit(10).peek(System.out::println).sum()
2
4
8
16
32
64
128
256
512
1024
$5 ==> 2046
```

- To convert primitive stream to reference stream
- collect() requires reference stream to save the output result stream into another list. So, boxed() must be used.

```
jshell> IntStream.iterate(2, e -> e*2).limit(10).collect(Collectors.toList())
|  Error:
|  method collect in interface java.util.stream.IntStream cannot be applied to given types;
|     required: java.util.function.Supplier<R>,java.util.function.ObjIntConsumer<R>,java.util.function.BiConsumer<R,R>
|found: java.util.stream.Collector<java.lang.Object,capture#2 of ?,java.util.List<java.lang.Object>>
|     reason: cannot infer type-variable(s) R
|       actual and formal argument lists differ in length)
|  IntStream.iterate(2, e -> e*2).limit(10).collect(Collectors.toList())
|  ^---------------------------------------------^

jshell> IntStream.iterate(2, e -> e*2).limit(10).boxed().collect(Collectors.toList())
$6 ==> [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

- Big Number Calculations with BigInteger

```
jshell> Integer.MAX_VALUE // That means operations limit to this can be done
$7 ==> 2147483647

jshell> Long.MAX_VALUE // That means operations limit to this can be done
$8 ==> 9223372036854775807
```

```
jshell> IntStream.rangeClosed(1,50).reduce(1, (x,y) -> x*y)
$9 ==> 0  // Because the result is out of Int limit value
```

```
jshell> LongStream.rangeClosed(1,20).reduce(1, (x,y) -> x*y)
$10 ==> 2432902008176640000

jshell> LongStream.rangeClosed(1,50).reduce(1, (x,y) -> x*y)
```

```
5.  $10 ==> -3258495067890909184 // Negative because out of range
```

```
1.   jshell> LongStream.rangeClosed(1,50).mapToObj(BigInteger::valueOf).reduce(BigInteger.ONE,
     BigInteger::multiply)
2.  $12 ==> 30414093201713378043612608166064768844377641568960512000000000000
```

Unit 7: Playing further with Java Functional Programming

Task 14 – Joining Strings in streams

Joining Strings with joining and Playing with flatMap

joining()

- It takes separator as argument

```
jshell> List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices",
   "AWS", "PCF", "Azure", "Docker", "Kubernetes");
courses ==> [Spring, Spring Boot, API, Microservices, AWS, PCF, Azure, Docker, Kubernetes]

jshell> courses.stream().collect(Collectors.joining(" "));
$14 ==> "Spring Spring Boot API Microservices AWS PCF Azure Docker Kubernetes"

jshell> courses.stream().collect(Collectors.joining(","));
$15 ==> "Spring,Spring Boot,API,Microservices,AWS,PCF,Azure,Docker,Kubernetes"
```

split()

- Converts the string to character array

```
jshell> "Spring".split("")
$16 ==> String[6] { "S", "p", "r", "i", "n", "g" }
```

```
jshell> courses.stream().map(course -> course.split("")).collect(Collectors.toList());
$17 ==> [[Ljava.lang.String;@3108bc, [Ljava.lang.String;@370736d9,
   [Ljava.lang.String;@5f9d02cb, [Ljava.lang.String;@63753b6d, [Ljava.lang.String;@6b09bb57,
   [Ljava.lang.String;@6536e911, [Ljava.lang.String;@520a3426, [Ljava.lang.String;@18eed359,
   [Ljava.lang.String;@3e9b1010]
```

- From above code, map returns stream of string arrays whereas collect accepts stream of characters. To overcome this, we need to flatten the stream of string arrays into character stream with the help of flatMap.

flatMap()

- It flattens the data into characters

```
jshell> courses.stream().map(course ->
   course.split("")).flatMap(Arrays::stream).collect(Collectors.toList());
$18 ==> [S, p, r, i, n, g, S, p, r, i, n, g,  , B, o, o, t, A, P, I, M, i, c, r, o, s, e,
   r, v, i, c, e, s, A, W, S, P, C, F, A, z, u, r, e, D, o, c, k, e, r, K, u, b, e, r, n, e,
   t, e, s]
```

```
jshell> courses.stream().map(course ->
   course.split("")).flatMap(Arrays::stream).distinct().collect(Collectors.toList())
;
$19 ==> [S, p, r, i, n, g,  , B, o, t, A, P, I, M, c, s, e, v, W, C, F, z, u, D, k, K, b]
```

Author – Abhinav (GitHub - @abhinavg916)

Task 15 – Creating Pairs

```
1. List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices", "AWS",
   "PCF", "Azure", "Docker", "Kubernetes");
2. courses ==> [Spring, Spring Boot, API, Microservices, AWS, PCF, Azure, Docker, Kubernetes]
3.
4. jshell> List<String> courses2 = List.of("Spring", "Spring Boot", "API", "Microservices",
   "AWS", "PCF", " zure", "Docker", "Kubernetes");
5. courses2 ==> [Spring, Spring Boot, API, Microservices, AWS, PCF, Azure, Docker,
   Kubernetes]
6.
7. jshell> courses.stream().flatMap(course -> courses2.stream().map(course2 ->
   List.of(course, course2)))
8. $6 ==> java.util.stream.ReferencePipeline$7@4f7d0008
9.
10. jshell> courses.stream().flatMap(course -> courses2.stream().map(course2 ->List.of(course,
    course2))).collect(Collectors.toList())
11.  $7 ==> [[Spring, Spring], [Spring, Spring Boot], [Spring, API], [Spring, Microservices],
    [Spring, AWS], [Spring, PCF], [Spring, Azure], [Spring, Docker], [Spring, Kubernetes],
    [Spring Boot, Spring], [Spring Boot, Spring Boot], [Spring Boot, API], [Spring Boot,
    Microservices], [Spring Boot, AWS], [Spring Boot, PCF], [Spring Boot, Azure], [Spring
    Boot, Docker], [Spring Boot, Kubernetes], [API, Spring], [API, Spring Boot], [API, API],
    [API, Microservices], [API, AWS], [API, PCF], [API, Azure], [API, Docker], [API,
    Kubernetes], [Microservices, Spring], [Microservices, Spring Boot], [Microservices, API],
    [Microservices, Microservices], [Microservices, AWS], [Microse ... ], [Docker, API],
    [Docker, Microservices], [Docker, AWS], [Docker, PCF], [Docker, Azure], [Docker, Docker],
    [Docker, Kubernetes], [Kubernetes, Spring], [Kubernetes, Spring Boot], [Kubernetes, API],
    [Kubernetes, Microservices], [Kubernetes, AWS], [Kubernetes, PCF], [Kubernetes, Azure],
    [Kubernetes, Docker], [Kubernetes, Kubernetes]]
```

- Eliminating the same elements pairs like [Spring, Spring], etc.

```
1. jshell> courses.stream().flatMap(course -> courses2.stream().map(course2 ->List.of(course,
   course2))).filter(list -> !list.get(0).equals(list.get(1))).collect(Collectors.toList())
2. $8 ==> [[Spring, Spring Boot], [Spring, API], [Spring, Microservices], [Spring, AWS],
   [Spring, PCF], [Spring, Azure], [Spring, Docker], [Spring, Kubernetes], [Spring Boot,
   Spring], [Spring Boot, API], [Spring Boot, Microservices], [Spring Boot, AWS], [Spring
   Boot, PCF], [Spring Boot, Azure], [Spring Boot, Docker], [Spring Boot, Kubernetes], [API,
   Spring], [API, Spring Boot], [API, Microservices], [API, AWS], [API, PCF], [API, Azure],
   [API, Docker], [API, Kubernetes], [Microservices, Spring], [Microservices, Spring Boot],
   [Microservices, API], [Microservices, AWS], [Microservices, PCF], [Microservices, Azure],
   [Microservices, Docker], [Microservices, Kubernetes] ... tes], [Docker, Spring], [Docker,
   Spring Boot], [Docker, API], [Docker, Microservices], [Docker, AWS], [Docker, PCF],
   [Docker, Azure], [Docker, Kubernetes], [Kubernetes, Spring], [Kubernetes, Spring Boot],
   [Kubernetes, API], [Kubernetes, Microservices], [Kubernetes, AWS], [Kubernetes, PCF],
   [Kubernetes, Azure], [Kubernetes, Docker]]
```

- Pairs on the basis of their length

```
1.  jshell> courses.stream().flatMap(course -> courses2.stream().filter(course2 ->
   course2.length()==course.length()).map(course2 -> List.of(course, course2))).filter(list -
   > !list.get(0).equals(list.get(1))).collect(Collectors.toList())
2. $9 ==> [[Spring, Docker], [API, AWS], [API, PCF], [AWS, API], [AWS, PCF], [PCF, API],
   [PCF, AWS], [Docker, Spring]]
```

Creating Higher-Order Functions

- It is a function that returns a function

```
1. // int cutoffReviewScore = 95;
2. // Predicate<Courses> reviewScoreGreaterThan95Predicate = course ->
   course.getReviewScore() > cutoffReviewScore;
3. // Predicate<Courses> reviewScoreGreaterThan90Predicate = course ->
   course.getReviewScore() > cutoffReviewScore;
4.
5. Predicate<Courses> reviewScoreGreaterThan95Predicate =
   createPredicateWithCutoffReviewScore(95);
```

```
6.  Predicate<Courses> reviewScoreGreaterThan90Predicate =
    createPredicateWithCutoffReviewScore(90);
7.  }
8.  private static Predicate<Courses> createPredicateWithCutoffReviewScore(int
    cutoffReviewScore) {
9.    return course -> course.getReviewScore() > cutoffReviewScore; // Higher Order Function
    because it returns logic (Predicate) instead of a value
10.   }
```

- Behaviour Parameterization helps in sending logic as parameter to a function and HOF helps in returning logic as parameter. This means function has become the first class citizen and hence functional programming.

FP and Performance - Intermediate Stream Operations are Lazy

- Functional Programming approach is faster than traditional approach
- For Example: Printing the optimal course such that string length > 11, convert it to uppercase and find the very first course that satisfies these conditions
- Using functional programming

```
1.  jshell> courses.stream().filter(course ->
    course.length()>11).map(String::toUpperCase).findFirst()
2.  $10 ==> Optional[MICROSERVICES]
```

```
1.  jshell> courses.stream().peek(System.out::println).filter(course ->
    course.length()>11).map(String::toUpperCase).peek(System.out::println).findFirst(
    )
2.  Spring
3.  Spring Boot
4.  API
5.  Microservices
6.  MICROSERVICES
7.  $12 ==> Optional[MICROSERVICES]
```

- From above, Java does not see for rest of the courses because of upon getting the first element that satisfied the condition and then stops. Thus, faster where as traditional approach has to first traverse all, calculates length of each one of them and then find the first element.
- Steps in traditional approach: Stream -> Filter all -> Map all to Uppercase -> Find the first element
- Steps in functional programming: Stream -> Filter -> (If failed then check for new elements from stream)
  -> (If passed then) Map -> Gets the first element and stops
- Writing highly performance code is easy in functional programming
- Intermediate operations in streams are LAZY which means executed only when terminal operation is executed.
- Java executes complete set of functional programming code when it knows the expected result (terminal operation). It does not execute for result until terminal operation is there. However, it may reserve the space and compile the statement but will not yield the final result. To get the result, terminal operation is needed.

```
1.  jshell> courses.stream().peek(System.out::println).filter(course ->
    course.length()>11).map(String::toUpperCase).peek(System.out::println)
2.  $13 ==> java.util.stream.ReferencePipeline$11@148080bb // $13 is a variable that is made
    by JShell
3.
4.  jshell> $13.findFirst() // Now, statement is executed and returns the result
5.  Spring
6.  Spring Boot
7.  API
8.  Microservices
9.  MICROSERVICES
10.  $14 ==> Optional[MICROSERVICES]
```

Improving Performance with Parallelization of Streams

Author – Abhinav (GitHub - @abhinavg916)

```
long time = System.currentTimeMillis();
            // Sequential
            // System.out.println(LongStream.range(0, 1000000000).sum()); // 1601 ms

            // Parallelized
            System.out.println(LongStream.range(0, 1000000000).parallel().sum()); // 188 ms

            System.out.println(System.currentTimeMillis() - time);
```

- It is not possible to parallelize the structured due to its state.
- State refers to the reparative change in value of a variable and hence not possible to run on different CPU cores. So, java has to run the complete code on single core.
- For Example:

```
List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);
            int sum  = 0;
            for(int number:numbers)
                    sum+=number; // Value keeps on changing
System.out.println(sum); // Structured Code
System.out.println(numbers.stream().reduce(0 , Integer::sum)); // Sequential
System.out.println(numbers.stream().parallel().reduce(0 , Integer::sum)); // Parallelized
```

- If there were 200,000 numbers then for a dual core CPU, one of CPU core will process 100,000 and other the CPU core will process 100,000. Thus, Efficiency of functional programming is beneficial and saves time.

Unit 8: Functional Programming makes Java Easy

Task 16 – Uppercase all the content in the list

Modifying lists with replaceAll and removeIf

- Traditional approach would have looped for each element

replaceAll()

- It is used to replace all each element with an operation
- It accepts function as a parameter

```
jshell> List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices",
"AWS", "PCF", "Azure", "Docker",
 "Kubernetes");
courses ==> [Spring, Spring Boot, API, Microservices, AWS, PCF, Azure, Docker, Kubernetes]

jshell> courses.replaceAll( str -> str.toUpperCase());
|  Exception java.lang.UnsupportedOperationException
|        at ImmutableCollections.uoe (ImmutableCollections.java:73)
|        at ImmutableCollections$AbstractImmutableList.replaceAll
(ImmutableCollections.java:105)
|        at (#2:1)

jshell> List<String> modifiableCourses = new ArrayList(courses)
|  Warning:
|  unchecked call to ArrayList(java.util.Collection<? extends E>) as a member of the raw
type java.util.ArrayList
|  List<String> modifiableCourses = new ArrayList(courses);
|                                   ^--------------------^
|  Warning:
|  unchecked conversion
|    required: java.util.List<java.lang.String>
|    found:    java.util.ArrayList
|  List<String> modifiableCourses = new ArrayList(courses);
|                                   ^--------------------^
modifiableCourses ==> [Spring, Spring Boot, API, Microservices, AWS, PCF, Azure, Docker,
Kubernetes]

jshell> modifiableCourses.replaceAll( str -> str.toUpperCase());

jshell> modifiableCourses
modifiableCourses ==> [SPRING, SPRING BOOT, API, MICROSERVICES, AWS, PCF, AZURE, DOCKER,
KUBERNETES]
```

removeIf()

- It removes the element based on the logic operation
- It accepts function as parameter

```
jshell> modifiableCourses.removeIf( str -> str.length()<6);
$7 ==> true

jshell> modifiableCourses
modifiableCourses ==> [SPRING, SPRING BOOT, MICROSERVICES, DOCKER, KUBERNETES]
```

Playing with Files using Functional Programming

- Functional Programming makes files easier for processing.

```
Files.lines(Paths.get("file.txt"))
                    .map( str -> str.split(" "))
                    .flatMap(Arrays::stream)
                    .distinct()
                    .sorted()
                    .forEach(System.out::println);
```

- To get the all the files and folders in the root directory

Author – Abhinav (GitHub - @abhinavg916)

```
-   Files.list(Paths.get(".")).forEach(System.out::println);
```

- To get only the folders in the root directory

```
1.  Files.list(Paths.get("."))
2.             .filter(Files::isDirectory)
3.             .forEach(System.out::println);
```

Playing with Threads using Functional Programming

- Functional Programming makes easier for implementing threads
- Using Structured Approach

```
1.  Runnable runnable = new Runnable() {
2.                      @Override
3.                      public void run() {
4.                              for (int i = 0; i < 10; i++)
5.                      System.out.println(Thread.currentThread().getId() + ":" + i);
6.                      }
7.    };
8.              Thread thread = new Thread(runnable);
9.              thread.start();
10.
11.             Thread thread1 = new Thread(runnable);
12.             thread1.start();
13.
14.             Thread thread2 = new Thread(runnable);
15.             thread2.start();
```

- Using Functional Approach
- Runnable interface is a functional interface itself

```
1.  Runnable runnable2 = () -> { // Lambda Expression
2.              IntStream.range(0, 10)
3.              .forEach(i -> System.out.println(Thread.currentThread().getId() + ":" + i));
4.                      };
5.
6.              Thread thread = new Thread(runnable2);
7.              thread.start();
8.
9.              Thread thread1 = new Thread(runnable2);
10.             thread1.start();
11.
12.             Thread thread2 = new Thread(runnable2);
13.             thread2.start();
```

Using Functional Programming in Java Applications

- It is paradigm shift in programming world
- It requires learn and un-learning of concepts
- It is good to use in java applications but be aware about the colleagues in the organisation that they must know functional programming as well.
- It benefits in faster performance, code readability and reusability.