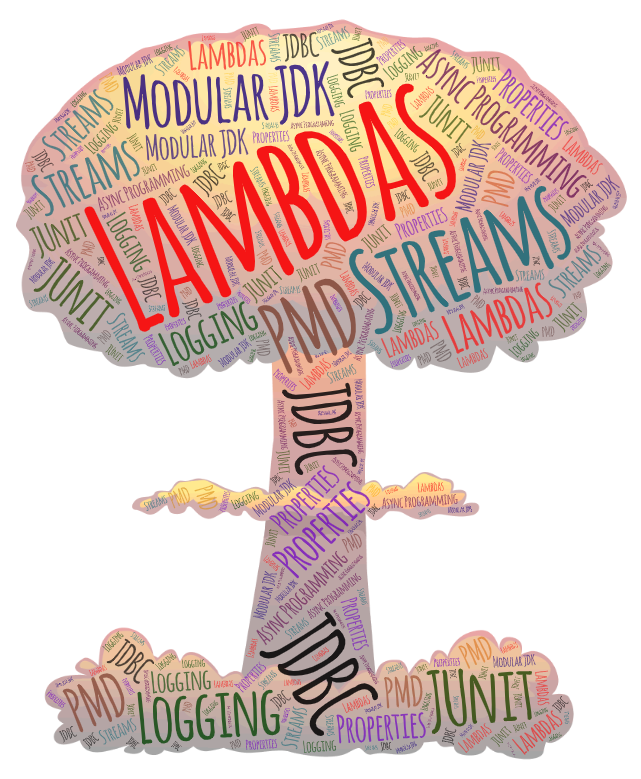
|  |
| --- |
| JAVA 11 |

Java 11 Beyond Essentials has been designed to enhance your knowledge of the Java 11 concepts. This course will take you through the advanced concepts in Java along with the usage of different tools in the development of an enterprise application such as logging, testing, etc.

Up until now, you have worked with the conditional and iterative statements to implement your business logic. Here, you will learn to work with Higher-Order functions like Lambdas and Streams which will simplify most of your business logic by the size and complexity. Scores of lines will get reduced to fewer lines, the code as a whole will become readable and the modularity also increases. But just making the code to implement to logic is not enough, you will also have to follow some coding standard, you will have to test the code you have written and also check if the testing has been done right. Along with development, we also have to take care of the maintenance of the project. This is possible when we log each process of the application.

With all these functionalities, will an external source be required? Yes. Here you will also learn the management of an external source or using a tool that will help you in it. Also, you will learn the why and how of persisting data to a central database. You will also find glimpses of concurrent programming, dive more into modularity and touch the surface of upholding your project using a tool called Docker.



Lambdas:

Josh and Remmy, two amateur coders, want to write a Java code for simple algebraic operations. Their requirement involves:

* Two number inputs on which the operations will be done.
* An input that specifies the type of operation to be done (assuming add, subtract, multiply and divide are the only options)
* A method *doOperation()*that processes the number inputs and the operation to de done and calls the appropriate method for the operation.

Remmy came up with a quick solution. Let's take a look at it:

1. *//Main Method*
2. public static void main(String[] args) {
4. System.out.println("Result is: " + doOperation(12, 4, "multiply"));
5. }

She called a function *doOperation()*which takes two numbers as inputs for operation, and a String input which specifies the type of operations to be done. This method also returns the double result of the operation, to be displayed in the console.

1. public static double doOperation(double num1, double num2, String operation) {
3. switch(operation) {
4. case "add":
5. return addCriteria(num1,num2);
6. case "subtract":
7. return subCriteria(num1,num2);
8. case "multiply":
9. return mulCriteria(num1,num2);
10. default:
11. *//Considering there are no wrong inputs, the default case is for operation=="divide"*
12. return divCriteria(num1,num2);
13. }
15. }

The *doOperation()*method checks the String argument passed and calls for the appropriate algebraic operation method, by passing the two numbers as arguments.

1. *//Method for Addition*
2. public static double addCriteria(double firstNum, double secondNum) {
3. double result = firstNum + secondNum;
4. return result;
5. }
6. *//Method for Subtraction*
7. public static double subCriteria(double firstNum, double secondNum) {
8. double result = firstNum - secondNum;
9. return result;
10. }
11. *//Method for Multiplication*
12. public static double mulCriteria(double firstNum, double secondNum) {
13. double result = firstNum \* secondNum;
14. return result;
15. }
16. *//Method for Division*
17. public static double divCriteria(double firstNum, double secondNum) {
18. double result = firstNum / secondNum;
19. return result;
20. }

Remmy also created four methods for individual algebraic operations, that takes two numbers as input, performs the operation and returns the result.

Josh liked the simplicity of Remmy's solution, but he also observed repetition and redundancy in the above code. Therefore, Josh wanted an even more compact solution. He used Lambda Expressions for his solution.

Now, before we look into Josh's solution to the above situation, let us learn about Lambda Expressions.

Lambda Expression is a type of "anonymous" method, which is not bound to any identifier. And so, they do not belong to any class/type. It brings a new and important feature to Java known as "Functional Programming". A Lambda Expression enables a method to be passed as an argument to other methods, as and when required.

Let's take a closer look at the following code for a better understanding.

1. @FunctionalInterface
2. interface Operation{
4. *//method providing definition of lambda expression*
5. public double calculations(double num1, double num2); *//Line 1*
6. }
7. public class Calculator {
8. public static void main(String[] args) {
10. *//creating lambda expression*
11. Operation adder = (double x, double y)-> x + y; *//Line 2*
13. *//executing lambda expression*
14. double result = adder.calculations(5, 6); *//Line 3*
16. System.out.println(result);
17. }
18. }

Here, we have a format for the Lambda Expressions at *Line 1,*which is derived from the method of Functional Interface *Operation*. The format defines Lambda Expression to be a method taking two double numbers and returning a double value. The actual lambda expression is created at *Line 2*, and it looks like:

1. (double x, double y) -> x + y *// Here "->" is the Lambda Operator used to define a Lambda Expression*

This Lambda Expression follows the format as derived from the Functional Interface. It takes two double arguments and returns the sum of the two double numbers. The Lambda Function is then assigned to an identifier, which has the type of the Functional Interface. The identifier and the functional method of interface is then used to access the Lambda Expression by passing relevant arguments.

Now that we have seen what Lambda Expression is, let's go deeper and see why should we use Lambda Expressions.

Let us consider the previous situation, and write the same functionality using a Class implementing the Functional Interface

1. *//Class*
2. class AddOperation implements Operation {
3. @Override
4. public double calculations(double num1, double num2) {
5. *//Returning the sum of two numbers*
6. return num1+num2;
7. }
8. }

And now, let's compare it with the code written using Lambda Expression.

1. *//Lambda Expression: Alternative to Regular Method*
2. Operation adder = (double num1, double num2)-> num1 + num2;

We can observe how compact the code written in Lambda Expression is, compared to that of a class. Let's observe how we can achieve the transition from a class to Lambda Expression.

The syntax for Lambda Expression is given by:

1. ( arguments ) -> { body }

The different parts of the syntax can be given as:

1. **Argument List**: The arguments in the regular method can be simply rewritten as:
   1. *//Arguments in Regular Method*
   2. public static double adder(double num1, double num2){}
   3. *//Argument List in Lambda Expressions*
   4. (double num1, double num2)

We can also observe that the Access Specifiers used in the regular method is dropped in Lambda Expression. This is because, Lambda Expression by definiton does not belong to any class. Futhermore, the Argument List should be exactly the same (in terms of number, type and order of arguments) as that of the abstract method in the interface.

* 1. *//Functional Interface method providing format for lambda expression*
  2. public double calculations(double num1,double num2);
  3. *//Lambda Expression created must follow the same format*
  4. Operation adder = (double num1, double num2) -> num1 + num2;

1. **Lambda Operator**: The arrow token "->" is used as the Lambda Operator to define a Lambda Expression. The Lambda Operator always comes after the argument list.
   1. *//Lambda Operator after the Argument List*
   2. (double num1, double num2) -> {body}

1. **Body**: The body of the regular method can be simply written after the Lambda Operator as
2. (double num1, double num2) -> {return num1 + num2}

    4. Since Java is intelligent enough to deduce the return type based on the input type, it is not necessary to mention either the return type of Lambda Expression or the return keyword. And since there is only one statement as the body for the Lambda Expression, the "{}" can be omitted as well. So, the above Lambda Expression can be rewritten as:

1. (double num1, double num2) -> num1 + num2

And thus, we obtain the Lambda Expression from the Regular Method for the given situation. Let's move on to the different styles of Lambda Expressions.

We have seen how streams can be used to perform operations such as filtering, sorting, etc. in a declarative way, making the syntax concise and easy to use.

However, with such operations at extensive levels comes the issue of performance and resource utilization. For example, operating on millions of records sequentially can degrade performance. Multithreading can provide a noticeable advantage here, but it has always been a challenging task for developers.

With the introduction of advanced concurrent programming features in Java, streams in the form of **parallel streams** not only utilize multithreading, but also provide a significant abstraction over thread pool management and the fork-join framework. This helps in utilizing the power of multicore CPUs for parallel processing in a simple declarative way.

Here is a familiar example using the parallelStream() method to retrieve a parallel stream from a collection:

1. public class FilterEmployee {
2. public static void main(String[] args) {
3. List<Employee> lstEmp = Employee.getEmpList();
4. lstEmp.parallelStream().filter(emp -> emp.getYearsInOrg() < 1)
5. .sorted((e1, e2) -> e1.getId() - e2.getId())
6. .forEach((e) -> System.out.println(e.getId() + ":" + e.getName()));
7. }
8. }

Parallel stream divides its elements into several chunks and processes each chunk on a different thread. By default, parallelStream() creates threads whose count equals the number of processors available.

Though parallel stream may look like an answer for faster performance, it may not always be the case. So benchmarking the performance and verifying the performance gain is an advisable step. Also checking the operations, which are being used in the stream can help in deciding when to use parallel streams. Operations which are dependent on the ordering of the elements like limit(), findFirst(), etc. are quite expensive in parallel stream.

Based on the number of arguments a Lambda Expression passes, there are three styles of writing Lambda Expressions:

* **Zero Parameter Lambda Expression**: A Lambda Expression containing no arguments.
  1. () -> System.out.println("No Arguments passed in this Lambda Expression")

* **One Parameter Lambda Expression**: A Lambda Expression containing one argument.
  1. *//For single parameter () is optional and declaration of type is optional*
  2. */\**
  3. *\* Note:*
  4. *\* For body consisting only 1 statement, {} is not necessary and*
  5. *\* Return keyword must not be used for returning values!*
  6. *\*/*
  7. num -> num+100
  8. *//If the data type of the single argument has to be declared, then the use of () is compulsory!!!!*
  9. */\*\**
  10. *\* Note:*
  11. *\* For body consisting more than 1 statements, {} is needed*
  12. *\* Return keyword is required to be used if any value is to be returned*
  13. *\*/*
  14. (double rate) -> { rate = rate\*100;
  15. System.out.println("One argument passed: " +rate);
  16. }

* **Multiple Parameter Lambda Expression**: A Lambda Expression containing two or more arguments.
  1. *//For multiple arguments, declaration is optional only when all arguments are of same type*
  2. *//Also the use of () is mandatory when declaring multiple arguments!!!*
  3. (num1, num2, num3) -> System.out.println("Multiple arguments passed: " +num1+","+num2+","+num3)
  4. (int num1, int num2, String result) -> {
  5. if(num1 + num2 > 100)
  6. result="Pass";
  7. else
  8. result="Fail";
  9. return result;
  10. }

Now, let's understand the Type of a Lambda Expression.

Now that we have seen what is Lambda Expression and it's different styles, a curiosity that pops up in our minds is: "What is the type of a Lambda Expression?

Are they objects of a class? Do they inherit from an inbuilt class?

In Java, Lambda Expressions are of the "target type" defined by Functional Interfaces. In simple words, they need to follow the definition of an abstract method defined inside the Functional Interface. Let us understand this with an example:

1. @FunctionalInterface
2. interface Operation{
3. *//Abstract Functional Method providing definition of lambda expression*
4. public double calculations(double num1,double num2);
5. }
6. *//Defining a lambda expression of type Operation*
7. Operation doAdd = (double x, double y)-> x+y;

Here, Lambda Expression has to follow the type defined in the interface. In the given example, the abstract method has

* A **Return** Type of **double**
* **Two Arguments** of **double** Type

Therefore, we cannot create the following Lambda Expression as it does not follow the definition of the Abstract Method:

1. *//Three input parameters cannot be passed*
2. Operation sub = (double x, double y , double z)-> (x+y-z)+"Hello"; *//String cannot be returned in place of double*

Hence, Lambda Expressions are a type of Functional Interface, where they follow the definition of the Abstract Functional Method of the interface.

Let us now look at the different types of Functional Interfaces using which Lambda Expressions are written.

Functional Interfaces can be classified into two types:

* **User-Defined Functional Interface**:  These are the Functional Interfaces that are defined by the programmer.
  1. *//User Defined Lambda Expression*
  2. @FunctionalInterface
  3. interface Operation{
  4. public double calculations(double num1,double num2);
  5. }

* **Built-In Functional Interface**: In order to reduce the frequent usage of User-Defined Functional Interfaces, Java provides a series of inbuilt functional interfaces. A lambda expression can be of any of these types. Some of them being:
  + Function – It represents a function that takes a single input parameter and returns a single value/object. For example:
    1. Function<Long, Long> addNum = (value) -> value + 10;
  + Predicate – It represents a function that takes a single value/object as a parameter, and returns true or false. For example:
    1. Predicate<Integer> checkAge = (age) -> age > 18;
  + Supplier – It represents a function that produces a value/an object without taking any input parameter. For example:
    1. Supplier<Integer> generateRandom = ()-> new Integer((int) (Math.random() \* 100));
  + Consumer – It represents a function that consumes or processes a value/an object without returning anything. For example:
    1. Consumer<String> printValue = (name)-> System.out.println(name);

For a detailed study on all the Inbuilt Functional Interfaces present in Java 11, please refer the following link:  [List of Functional Interfaces in Java 11](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html)

Now that we have studied what Lambda Expressions are, let's see why are Lambda Expressions important in the first place.

Now that we know about Lambda Expressions significantly, let's see what solution Josh came up with for The Calculator Scenario.

First, Josh came up with a user-defined functional interface Operation that defines the structure for the Lambda Expressions. He came up with the plan to write Lambda Expressions for each arithmetic operation, so opCriteria() the functional method of the interface defines the layout for the same. It defines the Lambda Expressions to take two numbers as inputs and return the double result as output.

1. interface Operation {
2. public double opCriteria(double firstNum, double secondNum);
3. }

Second, Josh came up with four Lambda Expressions for the four arithmetic operations. He assigned an identifier of the functional interface type to each of the Lambda Expression. He then invoked the method doOperation() that takes two number inputs and the identifier for the Lambda Expression required for operation. The doOperation() method returns a double result to be displayed in the console.

1. public static void main(String[] args) {
3. Operation add = (x,y) -> x+y;
4. Operation subtract = (x,y) -> x-y;
5. Operation multiply = (x,y) -> x\*y;
6. Operation divide = (x,y) -> x/y;

9. System.out.println("Result is: "+doOperation(12, 4, multiply));
10. }

At last, the doOperation() method simply invokes the functional method of the interface, which executes the Lambda Expression. And the result of the arithmetic operation is returned to the main() method for display.

1. public static double doOperation(double firstNum, double secondNum, Operation operator) {
2. return operator.opCriteria(firstNum, secondNum);
3. }

Isn't Josh's solution more concise, with respect to Remmy's code?

It certainly avoids the repetition and redundancy found in Remmy's solution. Also, it achieves the same result in 20-25 lines of code, as compared to 40-45 lines of code written in Remmy's solution. Apart from reducing the number lines of codes, Lambda Expressions are also helpful in reducing the conditional constructs (if-else constructs) used in the code. This can be observed in Josh's code, which makes zero usage of conditional constructs.

Hence, these are some of the major advantages of using Lambda Expressions over the normal procedural programming method.

Now, let's how can we make Lambda Expressions more concise.

Method Reference is an alternative to Lambda Expression. It is effective at situations where Lambda Expression is calling an existing method.

Syntax:

1. ClassName::methodName *// The "::" operator is used for Method Referencing*

Let’s consider the code given below:

1. List<String> strArr = List.of("Tyson", "Kai", "Max", "Ray", "Daichi");*//List.of() returns immutable list of String with mentioned values*
2. strArr.forEach(s -> System.out.println(s)); *//Printing List Using Lambda Expression*

Here, we have a Lambda Expression which helps us display the contents of the above-mentioned List. But what if we had a simpler solution to this Lambda Expression? Let's check the alternate code given below:

1. strArr.forEach(System.out::println); *//Printing List Using Method Reference*

Here, we can observe that both the code snippets work the same. Both of them execute the same logic. We can conclude that the second code is a concise and succinct version of the first code.

Method Reference helps in reducing lines of code, where Lambda Expression is calling an existing method. But it is not applicable in cases where we need to pass Arguments. This is because the feature of passing arguments is not supported by Method Reference.

The above example barely scratches the surface of Method References. There are more to the types of Method References and it's applications.

Now, we have successfully created the lambda expressions, but not used them yet. And in order to use them, we need to understand the concept of "Higher-Order Functions" first.

Now let see some of the best practices that can be followed while defining Lambda expressions are,

* Use of **@FunctionalInterface** annotation

A major requirement for any interface to be considered as a functional one is that there should be only one abstract method in it. But an interface can contain more than one abstract method, to make sure that not more than one abstract methods exist inside a functional interface it is recommended to use the **@FunctionalInterface**annotation. The compiler will throw an error whenever anyone tries to add a second abstract method to the functional interface.

Note:**@FunctionalInterface** is optional if the interface contains only one abstract method.

For Example:

1. *//Better way to use @Functional Interface*
2. @FunctionalInterface
3. interface MyInterface  {
4. void add(int num1, int num2);
5. }

* **Avoid parameter types while defining Lambda expressions**

The compiler will use the type inference to detect the data types of the parameters provided on its own in most cases, so avoid providing the datatype since it is optional, it keeps the code clean, compact and readable.

For Example:

1. (int num1, int num2) -> System.out.println(num1+ " " +num2);  *// Bad practice*
2. *//A parameter type is optional which can be avoided.*
3. (num1, num2) -> System.out.println(num1+ " " +num2);       *// Good practice*

* ***Braces{}* and *Return*statement are optional in one-line Lambda expressions**

As it was discussed already, Java is intelligent enough to deduce the return type based on the input type, it is not necessary to mention either the **return**type of Lambda Expression or the return keyword in a one-line lambda expression. And since there is only one statement as the body for the Lambda Expression, the **{ }**can be omitted as well

For Example:

1. str -> { return str.toUpperCase(); }  *// Bad practice*
2. str -> str.toUpperCase();   *// Good practice*

* **Avoid writing blocks of code in lambda expressions**

Lambda expressions should be short, concise, and clean. Writing a large block of code inside a lambda expression defeats the purpose of using lambda functions. So it is a good practice to not use lambda expressions whenever a large block of code has to be executed.

For Example:

Below example follows bad practice, a CustomerReport object is created for each iteration, all its values are set and then the object is added to another **returnList** object

1. *//Assume Customer, and CustomerReport exists*
2. *//Assume customerList contains all the Customer objects*
3. List<CustomerReport> returnList = new ArrayList<>(); *//list to populaate CustomerReport objects*
4. customerList.forEach(customer->{
5. CustomerReport customerReport = new CustomerReport();
6. customerReport.setCustomerId(customer.getCustomerId());
7. customerReport.setCustomerName(customer.getCustomerName());
8. customerReport.setEmailId(customer.getEmailId)):
9. *//code to set other required attributes*
10. returnList.add(customerReport);
11. });

The above code can be simplified as below

1. *//Assume Customer, and CustomerReport exists*
2. *//Assume customerList contains all the Customer objects*
3. List<CustomerReport> returnList = new ArrayList<>(); *//list to populaate CustomerReport objects*
4. customerList.forEach(customer->{
5. CustomerReport customerReport = new CustomerReport(customer.getCustomerId(),customer.getCustomerName(),customer.getEmailId));
6. returnList.add(customerReport);
7. });

In the above code, setter methods are replaced with a parameterized constructor to set the attributes. This makes the code much simpler, concise, and more readable. So, it is recommended to use lambda expressions when the block of code to be executed is small.

* **Prefer using Method references**

        As it was explained in the previous page, Java provides us with the ways to call existing methods in an easy and concise manner. It is preferred to use    these method references whenever possible to keep the code clean and concise

        For Example:

1. str -> str.toUpperCase();    *// Bad practice*
2. String::toUpperCase;     *// Good practice*

Problem Statement

Let us look at the Calculator Scenario once again. The requirements were:

* Two Number inputs on which the operations will be done.
* An input that specifies the type of operation to be done (assuming add, subtract, multiply and divide are the only options)
* A method doOperations() that processes the number inputs and the operation to de done, and calls the appropriate method for the operation.

Now let us look at Remmy's solution to this problem, which is coded without using Lambda Expression.

Problem Statement

Let us look at Josh's solution to the Calculator Scenario, which is created using Lambda Expression.

Problem Statement

Let's take a deeper look into Lambda Expressions as we observe the difference between Collections.sort() and list.sort().

Let us consider the following List of String:

List<String> nameList = Arrays.asList("Rachael","Ross","Monica","Chandler","Joey","Phoebe");

This list can be sorted in two ways:

* **Collections.sort(List<T> list)**: This method sorts elements in the given list in the ascending order, according to the natural order of the elements of the list. This method belongs to the class Collections and takes the List to be sorted
* **list.sort(Comparator<? super E> c)**: This method sorts elements in the given list according to the order mentioned by a Comparator. This method belongs to the interface List and takes a Lambda Expression for the customized sorting of the list. Hence, by this method we can sort a list not only on the basis of natural order, but also by our own custom format for sorting. We can now sort list based on length of each element, first character of each element etc.

So, we can observe how versatile sorting with list.sort() is in comparison to sorting with Collections.sort().

**Note**: Comparator is a Built-In Functional Interface, which is used in ordering a given collection. It orders the collection using the Lambda function provided to it.

For inducing ordering while sorting using Collections class, another overloaded method, sort(List<T> list, Comparator<? super T> c) can be used.

Problem Statement

Let's take a closer look at the Built-In Functional Interfaces provided by Java, and see how they work.

Different types of Built-In Functional Interfaces:

* Function<T1,T2>: Function takes only a single argument of type T1 and returns a value of type T2.
* Predicate<T>: Predicate takes only a single argument of type T and always returns a Boolean value.
* BiPredicate<T1,T2>: BiPredicate takes two arguments, each of type T1 and T2, and returns a Boolean value.
* Consumer<T>: Consumer consumes/takes only a single argument of type T, does operation/processing on it, and does not return any value.
* Supplier<T>: Supplier takes no arguments, and supplies/returns a value of type T.

Note: Since Java 11 var, the local variable type inference, can be used for parameter types in a Lambda expression. It will infer the type of the data based on the functional interfaces.

Problem Statement

Let us take alook at how objects of a User Defined Class can be passed as arguments to Built-In Functional Interfaces and Lambda Expressions with the following code.

**Note:** Comparator  (Comparable<T>) is a Built-In Functional Interface, which is used in ordering a given collection. It does the custom ordering of collections by taking Lambda Expressions as arguments, which provide the format for ordering.

Problem Statement

Problem Statement:

Consider the following Employee class:

class Employee {

public int id;

public String name;

private int salary;

private double yearsInOrg;

private String role;

private String gender;

public Employee(int id, String name, int salary, double years, String role, String gender) {

this.id = id;

this.name = name;

this.salary = salary;

this.yearsInOrg = years;

this.role = role;

this.gender = gender;

}

// Getters and setters

}

Harvey wants to display the name of all the employees and their gender. Help him in writing this code using lambda expressions.

.

Note: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

 Stream: (What is a higher order function?)

Functions that can either accept other functions as parameters or return other functions as parameters are called Higher Order Functions (HOF). It is an important part of the concept of Functional Programming.

Similar to Functional Interfaces, Higher-Order Functions can be either User-Defined or Inbuilt. Inbuilt Higher-Order Functions are available through Streams in Java 8 and onwards.

Let's take a closer look at Higher-Order Functions with the following examples:

1. List<String> stringList = new ArrayList<String>();
2. stringList.add("Abigail");
3. stringList.add("Zachariah");
4. stringList.add("Damien");
5. stringList.add("Ralph");
6. stringList.add("Gabriel");
7. stringList.sort((str1,str2)->str1.compareTo(str2));
8. System.out.println(stringList);

Here sort() is a Higher Order Function which is taking a Lambda Expression as an input parameter.

There are many more such high utility inbuilt higher-order functions available through Java Streams. Up next, let's learn what are Streams and how to create them.

A Stream represents a sequence of elements from a source and supports various data processing operations. In other words, it provides an abstraction over an existing collection. Streams also provide the support to parallel threading of the code, which breaks the bigger problem into smaller chunks. These smaller problems are solved separately and their solutions are combined together to obtain the final solution. This is possible, as Streams are both serially and parallelly executable.

Now, let's take a look at the different methods to create Built-In Streams in Java.

* Creating a Stream from a Collection using **stream()** method. For example:
  1. List<String> castList = List.of("Sam","Dean","Castiel","Crowley");
  2. Stream<String> supernatural = castList.stream();

* Creating a Stream from an Array using **stream()** method. For example:
  1. Integer[] array = {672, 340, 999};
  2. Stream<Integer> stream = Arrays.stream(array);

* Creating a Stream directly using **of()** method. For example:
  1. Creating a Stream directly for Integers:
     1. Stream<Integer> stream = Stream.of(672, 340, 999);
  2. Creating a Stream directly for Objects of User-Defined Classes:
     1. *//User Defined Class Employee*
     2. class Employee{
     3. private String name;
     4. private Double salary;
     5. public Employee(String name, Double salary) {
     6. super();
     7. this.name = name;
     8. this.salary = salary;
     9. }
     10. }
     11. *//Creating a Stream for objects of Class Employee*
     12. Stream<Employee> empStream = Stream.of( new Employee("Tom",5699.5),
     13. new Employee("Jack",7629.2),new Employee("Jane",5289.8));

Now that we have studied about Streams and the different ways to create them, let's see the different types of operations done on Streams.

The different operations that can be done on Streams are:

* **forEach()** : This method is used to traverse each element of the stream. It is used mainly to display the elements in a Stream. For example:
  1. List<String> placesToVisit= new ArrayList<String>();
  2. placesToVisit.add("Chicago");
  3. placesToVisit.add("Venice");
  4. placesToVisit.add("Tokyo");
  5. placesToVisit.add("San Francisco");
  6. placesToVisit.add("Kyoto");
  7. placesToVisit.add("Abu Dhabi");
  8. placesToVisit.forEach(place -> System.out.println("Trip to " + place));

* **map()** :  This method is used to return a new stream based on operations done on an existing stream. For example:
  1. List<String> placesToVisit= new ArrayList<String>();
  2. placesToVisit.add("Chicago");
  3. placesToVisit.add("Venice");
  4. placesToVisit.add("Tokyo");
  5. placesToVisit.add("San Francisco");
  6. placesToVisit.add("Kyoto");
  7. placesToVisit.add("Abu Dhabi");
  8. placesToVisit.stream().map(place -> place.toUpperCase()).forEach(place -> System.out.println(place));

* **filter()** : This method is used to return a new filtered stream based on conditions given for filtering. For example:
  1. List<String> placesToVisit= new ArrayList<String>();
  2. placesToVisit.add("Chicago");
  3. placesToVisit.add("Venice");
  4. placesToVisit.add("Tokyo");
  5. placesToVisit.add("San Francisco");
  6. placesToVisit.add("Kyoto");
  7. placesToVisit.add("Abu Dhabi");
  8. placesToVisit.stream().filter(place -> place.length() == 5).forEach(x -> System.out.println(x));

* **sorted()** : This method is used to sort the elements in a Stream. Argument for this method is optional. For Example:
  1. *//No Argument passed in sorted()*
  2. List<String> placesToVisit= new ArrayList<String>();
  3. placesToVisit.add("Chicago");
  4. placesToVisit.add("Venice");
  5. placesToVisit.add("Tokyo");
  6. placesToVisit.add("San Francisco");
  7. placesToVisit.add("Kyoto");
  8. placesToVisit.add("Abu Dhabi");
  9. placesToVisit.stream().sorted().forEach(x -> System.out.println(x));
  10. *//Lambda Expression passed as Argument in sorted()*
  11. List<String> placesToVisit= new ArrayList<String>();
  12. placesToVisit.add("Chicago");
  13. placesToVisit.add("Venice");
  14. placesToVisit.add("Tokyo");
  15. placesToVisit.add("San Francisco");
  16. placesToVisit.add("Kyoto");
  17. placesToVisit.add("Abu Dhabi");
  18. placesToVisit.stream().sorted((str1,str2) -> str1.compareTo(str2)).forEach(x -> System.out.println(x));

* **collect()** : This method stores the modified stream as a new collection type (it can be list, map etc.), after the stream operation terminates, under a new identifier.
  1. List<Integer> number = new ArrayList();
  2. number.add(2);
  3. number.add(3);
  4. number.add(4);
  5. number.add(5);
  6. *//The Modified Stream is stored in "doubled" using collect()*
  7. List<Integer> doubled = number.stream().map(x->2\*x).collect(Collectors.toList());
  9. System.out.println(doubled);

Now that we have observed the different types of operations done on Streams, let us now see how can we chain them together to obtain unique results.

The different operations of Streams, as we have studied before, can be chained together to obtain unique results on a Stream. Basically, we get ease to filter an existing Stream and display or store the modified/filtered Stream as per our requirements.

Let us consider the following code, where we apply the different operations of Stream individually:

1. List<String> placesToVisit= new ArrayList<String>();
2. placesToVisit.add("Chicago");
3. placesToVisit.add("Venice");
4. placesToVisit.add("Tokyo");
5. placesToVisit.add("San Francisco");
6. placesToVisit.add("Kyoto");
7. placesToVisit.add("Abu Dhabi");
9. *//stream() converts the List into a Stream*
10. Stream<String> streamPlaces = placesToVisit.stream();
12. *//filter() applies the appropriate filter conditions to obtain Modified Stream*
13. Stream<String> filterStream = streamPlaces.filter(place -> place.length() <= 8);
15. *//map() applies appropriate modification to a given Stream*
16. Stream<String> mapFilter = filterStream.map(place -> place.toUpperCase());
18. *//forEach() displays all the content of the Stream in console*
19. mapFilter.forEach(x -> System.out.println(x));

This code can also be shortened by chaining all the separate methods together, in the same order:

1. List<String> placesToVisit= new ArrayList<String>();
2. placesToVisit.add("Chicago");
3. placesToVisit.add("Venice");
4. placesToVisit.add("Tokyo");
5. placesToVisit.add("San Francisco");
6. placesToVisit.add("Kyoto");
7. placesToVisit.add("Abu Dhabi");
8. *//Chaining Multiple Operations of Streams Together*
9. placesToVisit.stream() *//stream()*
10. .filter(place -> place.length() <= 8) *//filter()*
11. .map(place -> place.toUpperCase()) *//map()*
12. .forEach(x -> System.out.println(x)); *//forEach()*

Both the codes, as we can see, give us the same output. Thus, we can observe how the chaining of different operations makes the code for filtering a List compact. And we can use different combinations of operations to obtain unique results.

It is always an efficient coding way to use the order of operations as a filter(), map(), sort(), and forEach() which improves the performance.

Let us now further look into the actual functioning of Chained Stream operations.

Now that we know about various operations of Streams and how to chain them together, let us know more about Stream Pipelining and Intermediate and Terminal operations.

Streams follow a pipeline format of execution. All the operations that are chained together form the intermediate operations. The last operation, after which the final result is obtained, forms the terminal operation. All the intermediate operations are executed simultaneously based on the stream and the required conditions.

Operations such as map(), filter(), sorted() comprise the intermediate operations, as these operations look after the filtering, modifying, and ordering aspects of the streams. Also, intermediate operations return a stream after its execution. Whereas, operations such as forEach() and/or collect() comprise the terminal operations, as these operations do not return any stream. Instead, they deliver the final stream by either by displaying them (forEach()) or by storing them in a collection (collect()).

Let us take a closer look at this with the following illustrative example:

1. Stream<Integer> intStream = Stream.of(10,2,7,5,6,5,8,11);

Let us consider a Stream of random integers. Our goal is to display a sorted stream consisting of cubes of all even integers in the given stream. We have the following chained stream operations as our solution.

1. intStream.filter(n -> { System.out.println("Filtering Current Element: "+n); return n % 2 == 0; })
2. .map(n -> { System.out.println("Mapping Current Element: "+n); return n \* n \* n; })
3. .sorted()
4. .forEach(n -> System.out.println(n));

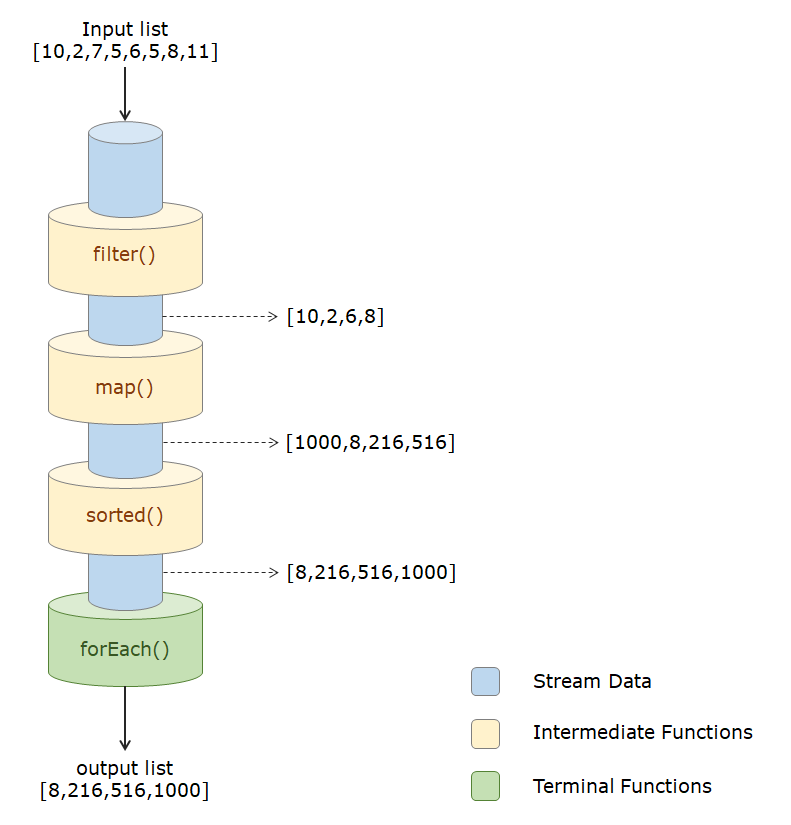
Let us take a look at the output before we analyze the pipeline execution of the given example.

1. Final stream is:
2. Filtering Current Element: 10
3. Mapping Current Element: 10
4. Filtering Current Element: 2
5. Mapping Current Element: 2
6. Filtering Current Element: 7
7. Filtering Current Element: 5
8. Filtering Current Element: 6
9. Mapping Current Element: 6
10. Filtering Current Element: 5
11. Filtering Current Element: 8
12. Mapping Current Element: 8
13. Filtering Current Element: 11
14. 8
15. 216
16. 512
17. 1000

We can observe that the first intermediate operation *filter()* executes for each element of the stream. After the execution of *filter(),*the next intermediate operation *map()* is executed for those elements which pass the filter condition. And after the execution of *map(),*the last intermediate operation *sorted()* and terminal operation *forEach()*is executed, where the filtered and mapped stream elements get sorted and displayed.

In the Pipelining of Streams, each operation in a new step improves readability.

This flow of execution can be visually represented as:



But pipelining of streams also result in Streams being lazy. Let us check how this happens up next.

In the previous example, we observed how the input stream got filtered, mapped and sorted before being displayed. We also observed how the even elements got filtered into a new stream, which then got modified (mapped to obtain cubes) and sorted.

But, what if we never invoke the terminal operation (in this case forEach()) after the intermediate operations? Will the elements of the stream still be filtered, mapped, and sorted?

1. Stream<Integer> intStream = Stream.of(10,2,7,5,6,5,8,11);
2. intStream.filter(n -> { System.out.println("Filtering Current Element: "+n); return n % 2 == 0; })
3. .map(n -> { System.out.println("Mapping Current Element: "+n); return n \* n \* n; })
4. .sorted();

And the answer is no.

This is because streams are lazy! Streams are said to be lazy because until the terminal operation is invoked, none of the intermediate operations is executed.

So, there is no terminal operation invoked, the filter operation and the mapping operation will not be executed. As a result, the stream remains unoperated. Hence, missing out on the terminal operation will not throw any error/exception but the chained stream operations will not be executed further leading to a few logical errors in the program.

Now, let us move on and look at the "non-reusability" nature of Streams up next.

Let us consider the following scenario:

1. Stream<String> streamNames = Stream.of("Deadpool", "Iron Man", "Spiderman", "Captain America", "Punisher", "Black Panther");
2. streamNames.filter(name -> name.length()<=10) *//Line1*
3. .sorted()
4. .forEach(name -> System.out.println(name));

We have created a Stream of string objects here. On this stream, we are applying a *filter()* and *sorted()* operation and displaying the modified stream using *forEach()*. Now let us try and reuse the same stream *streamNames* to map the elements of the stream using*map()* and displaying the modified stream again.

1. streamNames.map(name -> name.toUpperCase()) *//Line2*
2. .forEach(name -> System.out.println(name));

But upon doing so, a runtime exception will be thrown. This is because once a stream is operated on it cannot be operated on once more. Imagine streams to be like water pipes. Once the water flows from a pipe, there will be no water left. The same case applies to streams as well. So in the above situation, once the*filter()* operation ends at *Line1,*the stream *streamNames* becomes empty and does not allow any more operation to be invoked on it. And when the same stream is accessed again at *Line2*, it throws the following exception:

1. Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
2. at java.base/java.util.stream.AbstractPipeline.<init>(AbstractPipeline.java:203)
3. at java.base/java.util.stream.ReferencePipeline.<init>(ReferencePipeline.java:94)
4. at java.base/java.util.stream.ReferencePipeline$StatelessOp.<init>(ReferencePipeline.java:696)
5. at java.base/java.util.stream.ReferencePipeline$3.<init>(ReferencePipeline.java:189)
6. at java.base/java.util.stream.ReferencePipeline.map(ReferencePipeline.java:188)
7. at com.lambada.StreamDemo.main(StreamDemo.java:21)

Hence, we can observe the non-reusability nature of streams from the above situation.

Problem Statement

Let us see how the chaining of different Stream operations function with the following example.

You can also play around with the different stream operations.

Problem Statement

In the previous tryout, we saw the pipeline structure of Streams. And now let us see how Streams are lazy! Let us take a closer look at this with the following code.

Here, upon execution we observe that there is no output in the console. This is because, here the stream operation is missing a terminal operation (*forEach()*or*collect()*). And since the terminal operation is not invoked, neither of the intermediate operations are invoked.

This shows the laziness of Streams.

Problem Statement

Now let's take a look at Non-Reusability of Streams with the following code:

Here in Situation 1, we can observe that once the first Stream Operation is executed, the second Stream Operation throws an exception. This is because the first stream operation closes the stream after its execution gets complete at Line1.

Now in Situation 2, let us remove the*forEach()* of first stream operation (at Line2) and end the operation at *sorted()*. Doing so will make the stream lazy. But can we reuse the same stream again?

And the answer is no. We cannot reuse the same stream again regardless of it having terminal operation or not. This is because a stream cannot be reused again once any operation is performed on it.

Problem Statement

Problem Statement

In a lottery contest the company decided to give prize to those participants whose name started with “J” and has exactly 4 letters in it. Write a chained stream operations based code to find the name of winners.

Input : [“Jack”, “John”, “Garry”, “Jill”, “Jim”, “Frank”]

Output: [Jack, John, Jill]

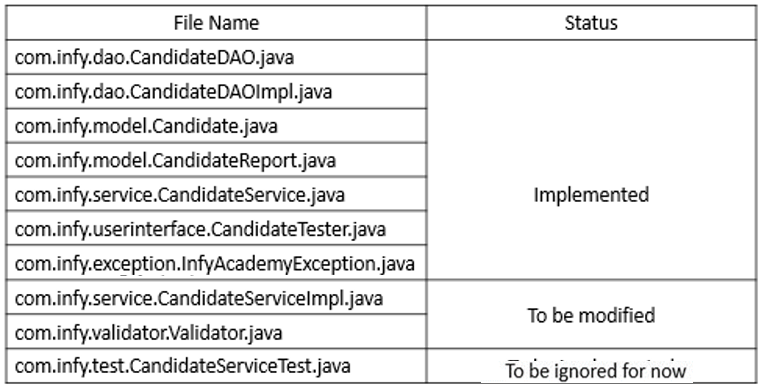
Note: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

Assignment on Stream:

XXXAcademy is an online academy that invites candidates to participate in various events, and also provides grades for each candidate based on his/her performance. They need your help in two modules,

* Adding candidates
* Getting candidate reports

**Artifacts:**



**Note: The above project is a maven project, no need to add external jars**

**Note: In this project properties are used, you will get about to know about  properties very soon**

**Exceptions**:

**InfyAcademyException**- (Implemented)

This is a user-defined Exception used to throw an exception in this application.

**InfyAcademyException**(String message)

This is a constructor for InfyAcademyException class with a message as a parameter.

**Validator:**

This class has been completely implemented for you. But we have used conditional statements everywhere. Please convert these statements to ternary operators or stream functions where ever possible.

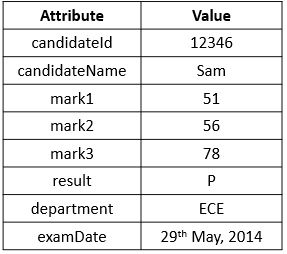
**CandidateServiceImpl:**

This class has been completely implemented for you. But we have used conditional statements and iterative statements. Please convert these statements to ternary operators or stream functions where ever possible.

**Sample Input and Output:**

**Add candidates:**

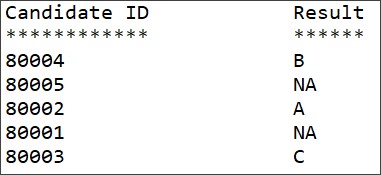
* I**nput:**



* **Output:**Candidate details are successfully added.

**Get Candidates Results:**

* **Output**



**Note**: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.