JAVA 8 FEATURES

Java 8 was released in 18th March 2014.

Some of the important Java 8 features are;

1. forEach() method in Iterable interface
2. Method references
3. Functional Interfaces
4. Lambda Expressions
5. default and static methods in Interfaces
6. Java Stream API for Bulk Data Operations on Collections
7. Java Time API
8. Collection API improvements
9. Concurrency API improvements
10. Java IO improvements
11. Miscellaneous Core API improvements

1. **forEach() method in Iterable interface:**

- **Signature: default void** forEach(Consumer<**super** T>action)
- *forEach* loop provides programmers with **a new, concise and interesting way for iterating over a collection**.
- Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.
- This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

Why forEach() was introduced?

Whenever we need to traverse through a Collection, we need to create an Iterator whose whole purpose is to iterate over and then we have business logic in a loop for each of the elements in the Collection. We might get ConcurrentModificationException if iterator is not used properly.

java.util.ConcurrentModificationException:

java.util.ConcurrentModificationException is a very common exception when working with java collection classes.

Java Collection classes are fail-fast, which means if the Collection will be changed while some thread is traversing over it using iterator, the *iterator.next()* will throw **ConcurrentModificationException**.

Concurrent modification exception can come in case of multithreaded as well as single threaded java programming environment.

Java 8 has **introduced *forEach* method in java.lang.Iterator interface** so that while writing code we focus on business logic only.

forEach method **takes java.util.function.Consumer object as argument**, so it helps in having our business logic at a separate location that we can reuse.

Example:

Consumer Implementation:

```
public class ForEachExampleWithConsumerClass {
	public static void main(String[] args) {

		//Creating Collection
		List<Integer> myList = new ArrayList<>();
		for(int i = 0; i < 5; i++)
			myList.add(i+1);

		//traversing using Iterator
		Iterator<Integer> it = myList.iterator();
		while(it.hasNext()) {
			Integer i = it.next();
			System.out.println("Iterator Value::" + i);
		}

		//traversing through forEach method of Iterable with anonymous
class
		myList.forEach(new Consumer<Integer>() {

			public void accept(Integer i) {
				System.out.println("forEach anonymous class Value::"
+ i);
			}
		});

		//traversing with Consumer interface implementation
		MyConsumer action = new MyConsumer();
		myList.forEach(action);

	}
}

//Consumer implementation that can be reused
class MyConsumer implements Consumer<Integer> {

	public void accept(Integer i) {
		System.out.println("Consumer impl Value::"+ i);
	}

}
```

Lambda Implementation:

```
public class ForEachExampleWithLambda {

	public static void main(String[] args) {

		List<String> gamesList = new ArrayList<>();
		gamesList.add("Football");
		gamesList.add("Chess");
		gamesList.add("Hockey");

		System.out.println("Iterating by passing Lambda Expression");
		gamesList.forEach(game -> System.out.println(game));

		System.out.println();
		System.out.println("Iterating by passing Method Reference");
		gamesList.forEach(System.out::println);

	}
}
```

2. Method Reference:

- Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.
- Method references help to point to methods by their names. A method reference is described using "::" symbol.

Types of Method References

There are following types of method references in java:

1. Reference to a static method. **Syntax**: **ContainingClass::staticMethodName**
2. Reference to an instance method. **Syntax**: **containingObject::instanceMethodName**
3. Reference to a constructor. **Syntax**: **ClassName::new**

3. Functional Interfaces:

- An interface with **exactly one abstract method** becomes Functional Interface.
- It **can have any number of default and static methods**. It can also declare methods of Object class.
- Functional interfaces are also known as **Single Abstract Method Interfaces (SAM Interfaces**).
- We don't need to use @FunctionalInterface annotation to mark an interface as Functional Interface.
- @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interface.
- It helps to achieve functional programming approach.
- **java.lang.Runnable with single abstract method run()** is a great example of functional interface.

Examples:

```
@FunctionalInterface
interface Sayable {
	void say(String msg);
}

public class FunctionalInterfaceExample implements Sayable {

	@Override
	public void say(String msg) {
		System.out.println(msg);
	}

	public static void main(String[] args) {
		FunctionalInterfaceExample fie = new
FunctionalInterfaceExample();
		fie.say("Functional Interface Example");
	}
}
```

A functional interface can have methods of object class.

```
@FunctionalInterface
interface SayableWithMethods {
	void say(String msg);   //abstract message

	//Can contain any number of Object class methods.
	int hashCode();
	String toString();
	boolean equals(Object obj);
}
```

```
public class FunctionInterfaceWithMethods implements SayableWithMethods {
	public void say(String msg) {
		System.out.println(msg);
	}

	public static void main(String[] args) {
		FunctionInterfaceWithMethods fie = new
		FunctionInterfaceWithMethods();
		fie.say("Functional Interface can have methods of Object
class.");
	}

}
```

Reason:

The members of an interface are:

- Those members declared in the interface.
- Those members inherited from direct superinterfaces.

- **If an interface has no direct super interfaces, then the interface implicitly declares a public abstract member method m with signature s, return type r, and throws clause t corresponding to each public instance method m with signature s, return type r, and throws clause t declared in Object, unless a method with the same signature, same return type, and a compatible throws clause is explicitly declared by the interface.**

  It is a compile-time error if the interface explicitly declares such a method m in the case where m is declared to be final in Object.

  It follows that is a compile-time error if the interface declares a method with a signature that is override-equivalent to a public method of Object, but has a different return type or incompatible throws clause.

The interface inherits, from the interfaces it extends, all members of those interfaces, except for
(a) fields, classes, and interfaces that it hides and
(b) methods that it overrides.

Fields, methods, and member types of an interface type may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures. However, this is discouraged as a matter of style.

One of the major benefits of functional interface is the possibility to use **lambda expressions** to instantiate them. We can instantiate an interface with **anonymous class** but the code looks bulky.

```
public class FIWithAnnonymousClass {

	public static void main(String[] args) {
		Runnable r = new Runnable() {

			@Override
			public void run() {
				System.out.println("Run Method on Annonymous Class");
			}
		};
	}
}
```

Since functional interfaces have only one method, lambda expressions can easily provide the method implementation. We just need to provide method arguments and business logic.

For example, we can write above implementation using lambda expression as:

```
Runnable r = () -> {
			System.out.println("Runnable with Lambda Expression");
		};
```

So lambda expressions are means to create anonymous classes of functional interfaces easily. **There are no runtime benefits of using lambda expressions.**

A new package java.util.function has been added with bunch of functional interfaces to provide target types for lambda expressions and method references.

Invalid Functional Interface:

- A functional interface can extend another interface only when it does not have any abstract method**.**

Example:

```
interface Sayable1 {
    void say(String msg);    // abstract method
}

//Invalid '@FunctionalInterface' annotation; DoAble is not a functional interface
@FunctionalInterface
interface DoAble extends Sayable1 {
	void doIt();
}
```

4. **Lambda Expression:** Syntax: **(argument-list) -> {body}**

- Lambda expression helps us to write our code in functional style. It provides a clear and concise way to implement SAM interface (Single Abstract Method) by using an expression.
- It is very useful in collection library in which it helps to iterate, filter and extract data.
- Enables to treat functionality as a method argument, or code as data. A function that can be created without belonging to any class.
- It provides a clear and **concise way to represent one method interface using an expression**.
- It is very useful in collection library. It helps to iterate, filter and extract data from collection.
- The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code.
- In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.
- **Java lambda expression is treated as a function, so compiler does not create .class file**.
- Lambda expression **provides implementation of *functional interface***.
- **Functional Interface**: An interface which has only one abstract method is called functional interface. Java provides an annotation *@FunctionalInterface*, which is used to declare an interface as functional interface.
- Java language provide support for **using lambda expressions only with functional interfaces**.
- Lambda Expression provides reduced Lines of Code and Sequential and Parallel Execution Support.

Lambda expression:

- Can define anonymous functions
- Can be assigned to a variable
- Can be passed to functions
- Can be returned from functions

Java lambda expression is consisted of three components.

1. **Argument-list:** It can be empty or non-empty as well.
2. **Arrow-token:** It is used to link arguments-list and body of expression.
3. **Body:** It contains expressions and statements for lambda expression.

Example:

Write below anonymous Runnable using lambda expression:

```
Runnable r = new Runnable() {

                    @Override
                    public void run() {
                          System.out.println("Run Method on Annonymous Class");
                    }
              };
```

Lambda Expression:

```
Runnable r1 = () -> System.out.println("My Runnable");
```

Let's try to understand what is happening in the lambda expression above.

- Runnable is a functional interface, that's why we can use lambda expression to create its instance.
- Since run() method takes no argument, our lambda expression also have no argument.
- Just like if-else blocks, we can avoid curly braces ({}) since we have a single statement in the method body. For multiple statements, we would have to use curly braces like any other methods.

Example:

```
@FunctionalInterface
interface Drawable {
      void draw();
}
```

Implementing an interface method without using lambda expression.

```
public class ExampleWithoutLambda {
      public static void main(String[] args) {

              int width = 10;
              //without lambda, Drawable implementation using anonymous class
              Drawable d = new Drawable() {

                    @Override
                    public void draw() {
                          System.out.println("Implementing Drawable : width ::
" + width);
                    }
              };

              d.draw();
      }
}
```

Implementing the above example with the help of lambda expression.

```
public class LambdaExample {
	public static void main(String[] args) {
		int width = 5;

		//With lambda
		Drawable d = () -> {
			System.out.println("Drawing : width :: " + width);
		};
		d.draw();
	}
}
```

Examples w.r.t number of arguments:

A lambda expression can have zero or any number of arguments.

Lambda Expression: No Parameter

```
interface Sayable{
	public String say();
}

public class LambdaWithNoParam {
	public static void main(String[] args) {

		//Lambda Expression with No Parameter
		Sayable s = () -> {
			return "I have nothing to say.";
		};
		System.out.println(s.say());
	}
}
```

Lambda Expression: Single Parameter

```
interface SayableMsg {
	String say(String msg);
}

public class LambdaWithSingleParam {
	public static void main(String[] args) {

		//Lambda Expression with Single Parameter
		SayableMsg sayMsg = (name) -> {
			return "Hello" + name;
		};
		System.out.println(sayMsg.say("Java"));

		//Paranthesis can be ommitted
		SayableMsg sm = name -> {
			return "Hello" + name;
		};
		System.out.println(sm.say("Java"));
	}
}
```

Lambda Expression: Multiple Parameters

```
interface Addition {
	int add(int a, int b);
}

public class LambdaWithMultipleParam {
	public static void main(String[] args) {

		//Lambda expression with Multiple Parameters
		Addition adding = (a,b)-> { return a+b;};
		System.out.println(adding.add(100, 2));
	}
}
```

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

```
Addition a1 = (a, b) -> a + b;
System.out.println(a1.add(1, 2));
```

Powerful Comparison with Lambda Expression:

Basic Sort: Without Lambda:

```
new Comparator<Employee>() {
                    @Override
                    public int compare(Employee e1, Employee e2) {
                          return e1.getName().compareTo(e2.getName());
                    }
              };
```

In case of list,

```
Collections.sort(employees, new Comparator<Employee>() {
                    @Override
                    public int compare(Employee e1, Employee e2) {
                          return e1.getName().compareTo(e2.getName());
                    }
              });
```

Basic Sort: With Lambda:

We can bypass the anonymous inner class and achieve the same result with **simple, functional semantics**:

```
(final Employee e1, final Employee e2) ->
e1.getName().compareTo(e2.getName());
```

In case of List,

```
employees.sort(
      (Employee e1, Employee e2) -> e1.getName().compareTo(e2.getName()));
```

Note:

We are using **the new *sort* API added to *java.util.List* in Java 8** – instead of the old *Collections.sort* API.

We can further simplify the expression by not specifying the type definitions – **the compiler is capable of inferring these** on its own:

```
(e1, e2) -> e1.getName().compareTo(e2.getName())
```

Exceptions in Java 8 Lambda Expression :

In Java 8, Lambda Expressions started to facilitate functional programming by providing a concise way to express behavior. However, the *Functional Interfaces* provided by the JDK don't deal with exceptions very well.

1. **Handling UnChecked Exceptions:**

   Problem:

   We have a *List<Integer>* and we want to divide a constant, say 50 with every element of this list and print the results:

```
List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);
integers.forEach(i -> System.out.println(50 / i));
```

   This expression works but there's one problem. If any of the elements in the list is *0*, then we get an *ArithmeticException: / by zero*.

   Let's fix that by using a traditional *try-catch* block such that we log any such exception and continue execution for next elements:

```
integers.forEach(i -> {
                try {
                      System.out.println(50 / i);
                } catch(ArithmeticException e) {
                      System.out.println("Error Occurred : " +
                      e.getMessage());
                }
         });
```

   The use of *try-catch* solves the problem, but the conciseness of a *Lambda Expression* is lost and it's no longer a small function as it's supposed to be.

   Solution:

   To deal with this problem, we can write **a lambda wrapper for the lambda function**.Let's look at the code to see how it works:

```
static Consumer<Integer> lambdaWrapper(Consumer<Integer> consumer) {
    return i -> {
        try {
            consumer.accept(i);
        } catch (ArithmeticException e) {
            System.err.println(
              "Arithmetic Exception occured : " + e.getMessage());
        }
    };
}
List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);
integers.forEach(lambdaWrapper(i -> System.out.println(50 / i)));
```

Explanation:

At first, we wrote a wrapper method that will be responsible for handling the exception and then passed the lambda expression as a parameter to this method.

Observation:

The wrapper method works as expected but, it's basically removing the *try-catch* block from lambda expression and moving it to another method and it doesn't reduce the actual number of lines of code being written.

This is true in this case where the wrapper is specific to a particular use case but we can make use of generics to improve this method and use it for a variety of other scenarios:

```
static <T, E extends Exception> Consumer<T> consumerWrapper(Consumer<T>
consumer, Class<E> clazz) {

        return i -> {
              try {
                    consumer.accept(i);
              } catch (Exception ex) {
                    try {
                          E exCast = clazz.cast(ex);
                          System.err.println("Exception occured : " +
                          exCast.getMessage());
                    } catch (ClassCastException ccEx) {
                          throw ex;
                    }
              }
        };
  }


integers.forEach(consumerWrapper(
                    i -> System.out.println(50 / i),
                          ArithmeticException.class));
```

As we can see, this iteration of our wrapper method takes **two arguments, the lambda expression and the type of *Exception* to be caught**. This lambda wrapper is capable of handling all data types, not just *Integers*, and catch any specific type of exception and not the superclass *Exception*.

Also, notice that we have changed the name of the method from *lambdaWrapper* to *consumerWrapper*. It's because this method only handles lambda expressions for *Functional Interface* of type *Consumer*. We can write similar wrapper methods for other Functional Interfaces like *Function*, *BiFunction*, *BiConsumer* and so on.

2. **Handling Checked Exceptions:**

Problem:

We have a *List<Integer>* and we we want to write them to a file. This operation of writing to a file throws *IOException*.

```
List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);
integers.forEach(i -> writeToFile(i));
```

On compilation, we get the following error:

Unhandled exception type IOException

Since *IOException* is a checked exception, it must be handled.

Now there are two options, we may want to throw the exception and handle it somewhere else or handle it inside the method that has the lambda expression.

i. Throwing Checked Exception from Lambda Expressions:

Let's throw the exception from the method in which the lambda expression is written, in this case, the *main*:

```
public static void main(String[] args) throws IOException {

        List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);
        integers.forEach(i -> writeToFile(i));
}
```

Still, while compiling, we get the same error of unhandled *IOException*. This is because lambda expressions are similar to Anonymous Inner Classes.

In this case, the lambda expression is an implementation of *accept(T t)* method from *Consumer<T>* interface.

Throwing the exception from *main* is does nothing and since the method in the parent interface doesn't throw any exception, it can't in its implementation.

```
Consumer<Integer> consumer = new Consumer<Integer>() {
        @Override
        public void accept(Integer integer) throws Exception {
              writeToFile(integer);
        }
  };
```

The above code doesn't compile because the implementation of *accept* method can't throw any Exception.

The most straightforward way would be to use a *try-catch* and wrap the checked exception into an unchecked exception and rethrow:

```
integers.forEach(i -> {
                    try {
                          writeToFile(i);
                    } catch (IOException e) {
                          throw new RuntimeException(e);
                    }
              });
```

This approach gets the code to compile and run but has the same problem as the example in the case of unchecked exceptions in the previous section.

Since we just want to throw the exception, we need to write our own Consumer Functional Interface which can throw an exception and then a wrapper method using it.

Let's call it *ThrowingConsumer:*

```
@FunctionalInterface
public interface ThrowingConsumer<T, E extends Exception> {
    void accept(T t) throws E;
}


static <T> Consumer<T> throwingConsumerWrapper(
                    ThrowingConsumer<T, Exception> throwingConsumer) {

              return i -> {
                    try {
                          throwingConsumer.accept(i);
                    } catch (Exception ex) {
                          throw new RuntimeException(ex);
                    }
              };
      }
```

Now we can write our lambda expression which can throw exceptions without losing the conciseness.

```
integers.forEach(throwingConsumerWrapper(i -> writeToFile(i)));
OR
integers.forEach(throwingConsumerWrapper(ClassName::writeToFile));
```

ii. Handling a Checked Exception from Lambda Expressions:

In this final section, we will modify the wrapper to handle checked exceptions. Since our *ThrowingConsumer* interface uses generics, we can handle any specific exception.

```
static <T, E extends Exception> Consumer<T> handlingConsumerWrapper(
            ThrowingConsumer<T, E> throwingConsumer, Class<E>
exceptionClass) {

      return i -> {
            try {
                  throwingConsumer.accept(i);
            } catch (Exception ex) {
                  try {
                        E exCast = exceptionClass.cast(ex);
                        System.err.println("Exception occured : " +
exCast.getMessage());
                  } catch (ClassCastException ccEx) {
                        throw new RuntimeException(ex);
                  }
            }
      };
  }
```

We can user this wrapper in our example to handle only the *IOException* and throw any other checked exception by wrapping them in an unchecked exception:

```
List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);

integers.forEach(handlingConsumerWrapper(i -> writeToFile(i),
IOException.class));
```

OR

```
integers.forEach(handlingConsumerWrapper(CheckedExceptionSolution::writeT
oFile, IOException.class));
```

Similar to the case of unchecked exceptions, throwing siblings for other Functional Interfaces like *ThowingFunction*, *ThrowingBiFunction*, *ThrowingBiConsumer* etc. can be written along with their corresponding wrapper methods.

Lambda Expressions/Functional Interfaces: Best Practices:

- **Prefer Standard Functional Interfaces:**
  Functional interfaces, which are gathered in the **java.util.function** package, satisfy most developers' needs in providing target types for lambda expressions and method references.

  Each of these interfaces is general and abstract, making them easy to adapt to almost any lambda expression.

  Developers should explore this package before creating new functional interfaces.

- **Use the** *@FunctionalInterface* **Annotation:**
  Annotate your functional interfaces with *@FunctionalInterface.* At first, this annotation seems to be useless. Even without it, your interface will be treated as functional as long as it has just one abstract method.

  But imagine a big project with several interfaces – it's hard to control everything manually. An interface, which was designed to be functional, could accidentally be changed by adding of other abstract method/methods, rendering it unusable as a functional interface.

  But using the *@FunctionalInterface* annotation, the compiler will trigger an error in response to any attempt to break the predefined structure of a functional interface. It is also a very handy tool to make your application architecture easier to understand for other developers.

- **Don't Overuse Default Methods in Functional Interfaces:**
  You can easily add default methods to the functional interface. This is acceptable to the functional interface contract as long as there is only one abstract method declaration:

```
@FunctionalInterface
public interface Foo {
    String method();
    default void defaultMethod() {}
}
```

  Functional interfaces can be extended by other functional interfaces if their abstract methods have the same signature. For example:

```
@FunctionalInterface
public interface FooExtended extends Baz, Bar {}

@FunctionalInterface
public interface Baz {
    String method();
    default void defaultBaz() {}
}
```

```
@FunctionalInterface
public interface Bar {
    String method();
    default void defaultBar() {}
}
```

Just as with regular interfaces, extending different functional interfaces with the same default method can be problematic.
For example, assume that interfaces *Bar* and *Baz* both have a default method *defaultCommon()*. In this case, you will get a compile-time error:

```
interface Foo inherits unrelated defaults for defaultCommon() from types
Baz and Bar...
```

To fix this, *defaultCommon()* method should be overridden in the *Foo* interface. You can, of course, provide a custom implementation of this method. But if you want to use one of the parent interfaces' implementations (for example, from the *Baz* interface), add following line of code to the *defaultCommon()* method's body:

```
Baz.super.defaultCommon();
```

But be careful. **Adding too many default methods to the interface is not a very good architectural decision.** It is should be viewed as a compromise, only to be used when required, for upgrading existing interfaces without breaking backward compatibility.

- **Instantiate Functional Interfaces with Lambda Expressions:**
  The compiler will allow you to use an inner class to instantiate a functional interface. However, this can lead to very verbose code.

  You should prefer lambda expressions:

  ```
  Foo foo = parameter -> parameter + " from Foo";
  ```

  over an inner class:

  ```
  Foo fooByIC = new Foo() {
      @Override
      public String method(String string) {
          return string + " from Foo";
      }
  };
  ```

  The lambda expression approach can be used for any suitable interface from old libraries. It is usable for interfaces like *Runnable*, *Comparator*, and so on. **However, this doesn't mean that you should review your whole older codebase and change everything.**

- **Avoid Overloading Methods with Functional Interfaces as Parameters:**
  Use methods with different names to avoid collisions; let's look at an example:

```
public interface Adder {
    String add(Function<String, String> f);
    void add(Consumer<Integer> f);
}

public class AdderImpl implements Adder {

    @Override
    public  String add(Function<String, String> str) {
        return str.apply("Something ");
    }

    @Override
    public void add(Consumer<Integer> i) {}
}
```

  At first glance, this seems reasonable. But any attempt to execute any of *AdderImpl's* methods:

```
String concatStr = adderImpl.add(a -> a + " from lambda");
```

  ends with an error with the following message:

```
reference to add is ambiguous both method
add(java.util.function.Function<java.lang.String,java.lang.String>)
in AdderImpl and method add(java.util.function.Consumer<java.lang.Integer>)
in AdderImpl match
```

  To solve **this problem**, you have two options.
  The **first** is to use methods with different names:

```
String addWithFunction(Function<String, String> f);

void addWithConsumer(Consumer<Integer> f);
```

  The **second** is to perform casting manually. This is not preferred.

```
String r = Adder.add((Function) a -> a + " from lambda");
```

- **Don't Treat Lambda Expressions as Inner Classes:**

  Despite our previous example, where we essentially substituted inner class by a lambda expression, the two concepts are different in an important way: scope.

  When you use an inner class, it creates a new scope. You can overwrite local variables from the enclosing scope by instantiating new local variables with the same names. You can also use the keyword ***this*** inside your inner class as a reference to its instance.

  However, lambda expressions work with enclosing scope. You can't overwrite variables from the enclosing scope inside the lambda's body. In this case, the keyword *this* is a reference to an enclosing instance.

  For example, in the class `ScopeExperiment` you have an instance variable *value:*

```
private String value = "Enclosing scope value";
```

  Then in some method of this class place the following code and execute this method.

```
public class ScopeExperiment {

   private String value = "Enclosing scope value";

   public static void main(String[] args) {
         System.out.println(new ScopeExperiment().scopeExperiment());
   }

   public String scopeExperiment() {

         Experiment exp = new Experiment() {
               String value = "Inner class value";

               @Override
               public String method(String str) {
                     return this.value;
               }
         };
         String resultInnerClass = exp.method("");

         Experiment expLambda = parameter -> {
               String value = "Lambda value";
               return this.value;
         };
         String resultLambda = expLambda.method("");

         return "Results: resultIC = " + resultInnerClass + ",
resultLambda = " + resultLambda;
   }
}
```

```
Output:
Results: resultIC = Inner class value, resultLambda = Enclosing scope
value
```

As you can see, by calling *this.value* in `resultInnerClass`, you can access a local variable from its instance. But in the case of the lambda, *this.value* call gives you access to the variable *value* which is defined in the `ScopeExperiment` class, but not to the variable *value* defined inside the lambda's body.

- **Keep Lambda Expressions Short and Self-explanatory:**
  If possible, use one line constructions instead of a large block of code.

  Remember, **lambdas should be an expression, not a narrative.** Despite its concise syntax, **lambdas should precisely express the functionality they provide.**

  This is mainly stylistic advice, as performance will not change drastically. In general, however, it is much easier to understand and to work with such code.

  This can be achieved in many ways –

1. **Avoid Blocks of Code in Lambda's Body:**

   In an ideal situation, lambdas should be written in one line of code.

   With this approach, the lambda is a self-explanatory construction, which declares what action should be executed with what data (in the case of lambdas with parameters).

   If you have a large block of code, the lambda's functionality is not immediately clear.

   With this in mind, do the following:

```
Foo foo = parameter -> buildString(parameter);

private String buildString(String parameter) {
    String result = "Something " + parameter;
    //many lines of code
    return result;
}
```

   Instead of,

```
Foo foo = parameter -> {
    String result = "Something " + parameter;
    //many lines of code
    return result;
};
```

However, please don't use this "one-line lambda" rule as dogma (a principle that cannot be denied)**. If you have two or three lines in lambda's definition, it may not be valuable to extract that code into another method.**

2. **Avoid Specifying Parameter Types**

A compiler in most cases is able to resolve the type of lambda parameters with the help of **type inference**. Therefore, adding a type to the parameters is optional and can be omitted.

Do this:

```
(a, b) -> a.toLowerCase() + b.toLowerCase();
```

Instead of this,

```
(String a, String b) -> a.toLowerCase() + b.toLowerCase();
```

3. **Avoid Parentheses around a Single Parameter**

Lambda syntax requires parentheses only around more than one parameter or when there is no parameter at all. That is why it is safe to make your code a little bit shorter and to exclude parentheses when there is only one parameter.

So, do this:

```
a -> a.toLowerCase();
```

instead of this:

```
(a) -> a.toLowerCase();
```

4. **Avoid Return Statement and Braces:**

Braces and ***return*** statements are optional in one-line lambda bodies. This means, that they can be omitted for clarity and conciseness.

Do this:

```
a -> a.toLowerCase();
```

instead of this:

```
a -> {return a.toLowerCase()};
```

5. **Use Method References:**

Very often, even in our previous examples, lambda expressions just call methods which are already implemented elsewhere. In this situation, it is very useful to use another Java 8 feature: **method references**.

So, the lambda expression:

`a -> a.toLowerCase();` could be substituted by: `String::toLowerCase;`

- **Use "Effectively Final" Variables:**

  Accessing a non-final variable inside lambda expressions will cause the compile-time error. **But it doesn't mean that you should mark every target variable as *final.***

  According to the "**effectively final**" concept, a compiler treats every variable as *final,* as long as it is assigned only once.

  It is safe to use such variables inside lambdas because the compiler will control their state and trigger a compile-time error immediately after any attempt to change them.

  For example, the following code will not compile:

```
public void method() {
    String localVariable = "Local";
    Foo foo = parameter -> {
        String localVariable = parameter;
        return localVariable;
    };
}
```

  The compiler will inform you that:

```
Variable 'localVariable' is already defined in the scope.
```

  This approach should simplify the process of making lambda execution thread-safe.

- **Protect Object Variables from Mutation:**

  One of the main purposes of lambdas is use in parallel computing – which means that they're really helpful when it comes to thread-safety.

  The "effectively final" paradigm helps a lot here, but not in every case.

  Lambdas can't change a value of an object from enclosing scope. But in the case of mutable object variables, a state could be changed inside lambda expressions.

  Consider the following code:

```
int[] total = new int[1];
Runnable r = () -> total[0]++;
r.run();
```

  This code is legal, as *total* variable remains "effectively final". But will the object it references to have the same state after execution of the lambda? No!

4. Default and Static Methods in Interfaces:

DEFAULT METHODS (Defender Methods):

- Java provides a **facility to create default methods inside the interface**. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

- From Java 8, interfaces are enhanced to have method with implementation. We can **use default and static keyword to create interfaces with method implementation**.

- The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

- It provides flexibility to allow Interface define implementation which will use as *default* in the situation where a concrete Class fails to provide an implementation for that method.

- We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code.

- **Why default method?**

  For example, if several classes such as A, B, C and D implements an interface XyzInterface then if we add a new method to the XyzInterface, we have to change the code in all the classes (A, B, C and D) that implements this interface.

  In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes.

  This is why in java 8, we have a new concept "default methods". These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

- For example, 'List' or 'Collection' interfaces do not have 'forEach' method declaration. **The forEach isn't declared by java.util.List nor the java.util.Collection interface yet**. Thus, adding such method will simply break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not implement the same.

- If you read forEach method details carefully, you will notice that it's defined in Iterable interface but we know that interfaces can't have method body. From Java 8, interfaces are enhanced to have method with implementation. We can use default and static keyword to create interfaces with method implementation. forEach method implementation in Iterable interface is:

```
default void forEach(Consumer<? super T> action) {

        Objects.requireNonNull(action);
        for (T t : this) {
              action.accept(t);
        }
}
```

Example:

```
public interface MyInterface {

      /* This is a default method so we need not
       * to implement this method in the implementation
       * classes
       */
      default void newMethod(){
            System.out.println("Newly added default method");
      }

      /* Already existing public and abstract method
       * We must need to implement this method in
       * implementation classes.
       */
      void existingMethod(String str);

      /*A default method cannot override a method from java.lang.Object */
      /*default String toString() {
            return "String";
      }*/

}
public class DefaultMethodExample implements MyInterface {

      // implementing abstract method
      public void existingMethod(String str){
            System.out.println("String is: "+str);
      }

      public static void main(String[] args) {
            DefaultMethodExample obj = new DefaultMethodExample();

            //calling the default method of interface
            obj.newMethod();
            //calling the abstract method of interface
            obj.existingMethod("Java 8 is easy to learn");
      }

}
```

- We know that Java doesn't provide multiple inheritance in Classes because it leads to **Diamond Problem**. So how it will be handled with interfaces now, since interfaces are now similar to abstract classes. The solution is that compiler will throw exception in this scenario and we will have to provide implementation logic in the class implementing the interfaces.

- The multiple inheritance problem can occur, when we have two interfaces with the default methods of same signature.

  Example:

```
public interface MyInterface2 {

   default void newMethod() {
         System.out.println("Newly added default method");
   }

   void display(String str);

}

public class MultipleDefaultExample implements MyInterface, MyInterface2 {

   // implementing abstract methods
   public void existingMethod(String str){
         System.out.println("String is: "+str);
   }

   public void disp(String str){
         System.out.println("String is: "+str);
   }

   public static void main(String[] args) {
         MultipleDefaultExample obj = new MultipleDefaultExample();

         //calling the default method of interface
         obj.newMethod();
   }
}
```

//Error: Duplicate default methods named newMethod with the parameters () and () are inherited from the types MyInterface2 and MyInterface

This is because we have the same method in both the interface and the compiler is not sure which method to be invoked.

To solve this problem, we can implement this method in the implementation class like this:

```
public class MultipleDefaultExample implements MyInterface, MyInterface2 {

	// implementing abstract methods
	public void existingMethod(String str){
		System.out.println("String is: "+str);
	}

	public void display(String str){
		System.out.println("String is: "+str);
	}

	//Implementation of duplicate default method
	@Override
	public void newMethod(){
		System.out.println("Implementation of default method");
	}

	public static void main(String[] args) {
		MultipleDefaultExample obj = new MultipleDefaultExample();

		//calling the default method of interface
		obj.newMethod();
	}

}
```

OR

```
	@Override
	public void newMethod() {
			MyInterface.super.newMethod();

	}
```

When we implement an interface that contains a default method, we can perform the following action:

- Not override the default method and will inherit the default method.
- Override the default method and will inherit the default method.
- Re-declare default method as abstract, which will force sub-classes to override it.

A default method cannot override a method from java.lang.Object.

```
default String toString() {
	return "String";

}
```

If any class in the hierarchy has a method with same signature, then default methods become irrelevant. **Since any class implementing an interface already has Object as superclass, if we have equals(), hashCode() default methods in interface, it will become irrelevant**. That's why for better clarity, interfaces are not allowed to have Object class default methods.

Important points about java interface default methods:

1. Java interface default methods help us in extending interfaces without having the fear of breaking implementation classes.

2. Java interface default methods has bridge down the differences between interfaces and abstract classes.

3. Java 8 interface default methods help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.

4. Java interface default methods help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.

5. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.

6. **If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it's because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That's why to avoid confusion, we can't have default methods that are overriding Object class methods.**

7. Java interface default methods are also referred to as **Defender** Methods or **Virtual extension** methods.

STATIC METHODS:

- Java interface static method is similar to default method except that we can't override them in the implementation classes.
- This feature helps us in avoiding undesired results in case of poor implementation in implementation classes.
- The static methods in interface are similar to default method so we need not to implement them in the implementation classes.
- We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes.

Example:

```
public interface MyData {

   default void print(String str) {
         if (!isNull(str))
               System.out.println("MyData Print::" + str);
   }

   static boolean isNull(String str) {
         System.out.println("Interface Null Check");

         return null != str && str.isEmpty();
   }

}


public class MyDataImpl implements MyData {

   public boolean isNull(String str) {
         System.out.println("Impl Null Check");
         return str == null;
   }

   public static void main(String args[]){
         MyDataImpl obj = new MyDataImpl();
         obj.print("");
         obj.isNull("abc");
   }

}
```

Note: isNull(String str) is a simple class method, it's not overriding the interface method. For example, if we will add @Override annotation to the isNull() method, it will result in compiler error.

Java interface static method is visible to interface methods only, if we remove the isNull() method from the MyDataImpl class, we won't be able to use it for the MyDataImpl object. However like other static methods, **we can use interface static methods using class name**. For example, a valid statement will be:

```
boolean result = MyData.isNull("abc");
```

Difference between Default Method and Regular Method:

- Default Method is different from the regular method in the sense that default method comes with default modifier.
- Additionally, methods in Classes can use and modify method arguments as well as the fields of their Class but default method, on the other hand, can only access its arguments as Interfaces do not have any state.

Java 8 – Abstract classes' vs interfaces:

With the introduction of default methods in interfaces, abstract classes are similar as interface in Java 8. However this is not entirely true, even though we can now have concrete methods (methods with body) in interfaces just like abstract class, this doesn't mean that they are same. There are still few differences between them:

- One of them is that **abstract class can have constructor** while in **interfaces cannot** have constructors.

- The purpose of **interface is to provide full abstraction**, while the purpose of **abstract class is to provide partial abstraction**. The interface is like a blueprint for your class, with the introduction of default methods you can simply say that we can add additional features in the interfaces without affecting the end user classes.

- Abstract class are more structured and can have a state associated with them. While in contrast, *default method* can be implemented only in the terms of invoking other Interface methods, with no reference to a particular implementation's state.

Hence, both are used for different purposes and choosing between two really depends on the scenario context.

Important points about java interface static method:

1. Java interface static method is part of interface, we can't use it for implementation class objects.

2. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.

3. Java interface static method helps us in providing security by not allowing implementation classes to override them.

4. We **can't define interface static method for Object class methods**, we will get compiler error as "**This static method cannot hide the instance method from Object**". This is because **it's not allowed in java, since Object is the base class for all the classes and we can't have one class level static method and another instance method with same signature.**

5. We can use java interface static methods to remove utility classes such as Collections and move all of its static methods to the corresponding interface that would be easy to find and use.

5. Java Stream API for Bulk Data Operations on Collections:

- A new java.util.stream has been added in Java 8 to perform filter/map/reduce like operations with the collection. This package consists of classes, interfaces and enum to allow functional-style operations on the elements.

- Stream API allows sequential as well as parallel execution. This is one of the best feature while working with Collections and usually with Big Data, as we need to filter out them based on some conditions.

- Collection interface has been extended with *stream()* and *parallelStream()* default methods to get the Stream for sequential and parallel execution.

- Most of the Java 8 Stream API method arguments are functional interfaces, so lambda expressions work very well with them.

Stream provides following features:

  - No storage. Stream **does not store elements**. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
  - Stream is **functional in nature**. Operations performed on a stream does not modify its source. For example, filtering a Stream obtained from a collection produces a **new Stream** without the filtered elements, rather than removing elements from the source collection.
  - **Laziness-seeking**. Stream is lazy and evaluates code only when required. **Java 8 Stream internal iteration principle** helps in achieving lazy-seeking in some of the stream operations. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization.

    For example, "find the first String with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.

  - Possibly unbounded. While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.

  - Consumable. The **elements** of a stream are **only visited once during the life of a stream**. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

Why Stream API is required?

Prior to Java 8, the approach to do it would be:

```
private static int sumIterator(List<Integer> list) {
            Iterator<Integer> it = list.iterator();
            int sum = 0;
            while (it.hasNext()) {
                  int num = it.next();
                  if (num > 10) {
                        sum += num;
                  }
            }
            return sum;
}
```

There are three major problems with the above approach:

1. We just want to know the sum of integers but we would also have to provide how the iteration will take place, this is also called **external iteration** because client program is handling the algorithm to iterate over the list.
2. The program is sequential in nature, there is no way we can do this in parallel easily.
3. There is a lot of code to do even a simple task.

To overcome all the above shortcomings, Java 8 Stream API was introduced. We can use Java Stream API to implement **internal iteration** which is better because java framework is in control of the iteration.

Internal iteration provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.

Let's see how we can write above logic in a single line statement using Java Streams:

```
private static int sumStream(List<Integer> list) {
            return list.stream().filter(i -> i > 10).mapToInt(i -> i).sum();

}
```

Note: Above program utilizes java framework iteration strategy, filtering and mapping methods and would increase efficiency.

How to work with Stream in Java?

As we have seen in the above example, the working of stream can be explained in three stages:

1. Create a stream.

2. Perform **intermediate operations** on the initial stream to transform it into another stream and so on, further intermediate operations. In the above example, the filter() operation is intermediate operation, there can be more than one intermediate operations.

3. Perform **terminal operation** on the final stream to get the result. In the above example, the sum() operation is terminal operation.

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

- **Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
- **Source** − Stream takes Collections, Arrays, or I/O resources as input source.
- **Aggregate operations** − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
- **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
- **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Core concepts of Java 8 Stream API:

1. **Collections and Java Stream:**

   A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated. Whereas a java Stream is a data structure that is computed on-demand.

   Java 8 Stream support sequential as well as parallel processing, parallel processing can be very helpful in achieving high performance for large collections.

   All the Java Stream API interfaces and classes are in the java.util.stream package. Since we can use primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time, there are specific classes for primitive types – IntStream, LongStream and DoubleStream.

2. **Functional Interfaces in Java 8 Stream:**

   Some of the commonly used functional interfaces in the Java 8 Stream API methods are:

   1. **Function and BiFunction**: Function represents a function (**method**) that **takes one type of argument and returns another type of argument.** Function<T, R> is the generic form where T is the type of the input to the function and R is the type of the result of the function.

      For handling primitive types, there are specific Function interfaces – ToIntFunction, ToLongFunction, ToDoubleFunction, ToIntBiFunction, ToLongBiFunction, ToDoubleBiFunction, LongToIntFunction, LongToDoubleFunction, IntToLongFunction, IntToDoubleFunction etc.

   Some of the Stream methods where Function or its primitive specialization is used are:

   - <R> Stream<R> map(Function<? super T, ? extends R> mapper)
   - IntStream mapToInt(ToIntFunction<? super T> mapper) – similarly for long and double returning primitive specific stream.
   - IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper) – similarly for long and double
   - <A> A[] toArray(IntFunction<A[]> generator)
   - <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

2. **Predicate and BiPredicate:** It represents a predicate (**condition**) against which elements of the **stream are tested**. This is used to filter elements from the java stream. Just like Function, there are primitive specific interfaces for int, long and double.

   Some of the Stream methods where Predicate or BiPredicate specializations are used are:

   - Stream<T> filter(Predicate<? super T> predicate)
   - boolean anyMatch(Predicate<? super T> predicate)
   - boolean allMatch(Predicate<? super T> predicate)
   - boolean noneMatch(Predicate<? super T> predicate)

3. **Consumer and BiConsumer**: It represents **an operation that accepts a single input argument and returns no result.** It can be used to perform some action on all the elements of the java stream.

   Some of the Java 8 Stream methods where Consumer, BiConsumer or its primitive specialization interfaces are used are:

   - Stream<T> peek(Consumer<? super T> action)
   - void forEach(Consumer<? super T> action)
   - void forEachOrdered(Consumer<? super T> action)

4. **Supplier**: Supplier represent an operation through which we can generate new values in the stream.

   Some of the methods in Stream that takes Supplier argument are:

   - public static<T> Stream<T> generate(Supplier<T> s)
   - <R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T> accumulator,BiConsumer<R, R> combiner)

3. **java.util.Optional:**

   Java Optional is a container object which may or may not contain a non-null value.

   If a value is present, isPresent() will return true and get() will return the value.

   Stream terminal operations return Optional object.
   Some of these methods are:

   - Optional<T> reduce(BinaryOperator<T> accumulator)
   - Optional<T> min(Comparator<? super T> comparator)
   - Optional<T> max(Comparator<? super T> comparator)
   - Optional<T> findFirst()
   - Optional<T> findAny()

4. **java.util.Spliterator**
   For supporting parallel execution in Java 8 Stream API, Spliterator interface is used. **Spliterator trySplit()** method returns a new Spliterator that manages a subset of the elements of the original Spliterator.

Java Stream Intermediate and Terminal Operations

Java Stream API operations that returns a new Stream are called **intermediate** operations. Most of the times, these **operations are lazy in nature**, so they start producing new stream elements and send it to the next operation.

ntermediate operations are never the final result producing operations. Commonly used intermediate operations are filter and map.

Java 8 Stream API operations that returns a result or produce a side effect is called **terminal**. Once the terminal method is called on a stream, it consumes the stream and after that we can't use stream.

Terminal operations are eager in nature i.e they process all the elements in the stream before returning the result. Commonly used terminal methods are **forEach, toArray, min, max, findFirst, anyMatch, allMatch** etc. You can identify terminal methods from the return type, they will never return a Stream.

Creating streams using the existing data-provider sources:

Streams can be created from different element sources.
E.g. collection or array with the help of *stream()* and *of()* methods:

1. We can use ***Stream.of()*** to create a stream from similar type of data.
   For example, we can create Java Stream of integers from a group of int or Integer objects.

```
Stream<Integer> stream = Stream.of(1,2,3,4);
```

2. We can use Stream.of() with an array of Objects to return the stream.
   Note**: It doesn't support autoboxing**, so we can't pass primitive type array.

```
Stream<Integer> streamObj = Stream.of(new Integer[]{1,2,3,4});

Stream<Integer> stream1 = Stream.of(new int[]{1,2,3,4});
//Compile time error, Type mismatch: cannot convert from Stream<int[]> to Stream<Integer>
```

3. We can use Collection **stream**() to create sequential stream and **parallelStream()** to create parallel stream. A *stream()* default method is added to the *Collection* interface and allows creating a *Stream<T>* using any collection as an element source*:*

```
List<Integer> myList = new ArrayList<>();
for(int i = 0; i < 5; i++) myList.add(i);

//sequential stream
Stream<Integer> sequentialStream = myList.stream();

//parallel stream
Stream<Integer> parallelStream = myList.parallelStream();
```

4. We can use **Stream.generate()** and **Stream.iterate()** methods to create Stream.

```
Stream<String> stream1 = Stream.generate(() -> {return "abc";});
Stream<String> stream2 = Stream.iterate("abc", (i) -> i);
```

5. Using **Arrays.stream()** and **String.chars()** methods.

```
LongStream is = Arrays.stream(new long[]{1,2,3,4});
IntStream is2 = "abc".chars();
```

Converting Java Stream to Collection or Array:

There are several ways through which we can get a Collection or Array from a java Stream.

1. We can use java Stream **collect()** method to get List, Map or Set from stream.

```
Stream<Integer> intStream = Stream.of(1,2,3,4);
List<Integer> intList = intStream.collect(Collectors.toList());
System.out.println(intList);      //prints [1, 2, 3, 4]

intStream = Stream.of(1,2,3,4); //stream is closed, so we need to create
it again
Map<Integer,Integer> intMap = intStream.collect(
                            Collectors.toMap(i -> i, i -> i+10));
System.out.println(intMap);      //prints {1=11, 2=12, 3=13, 4=14}
```

2. We can use stream **toArray()** method to create an array from the stream.

```
Stream<Integer> intStream1 = Stream.of(1,2,3,4);
Integer[] intArray = intStream.toArray(Integer[]::new);

System.out.println(Arrays.toString(intArray)); //prints [1, 2, 3, 4]
```

Java Stream Intermediate Operations:

- **Stream filter():**
  We can use filter() method to test stream elements for a condition and generate filtered list.
  Take arguments as **Predicate** (**boolean-valued function**).

- **Stream map():**
  We can use map() to apply functions to an stream.
  Takes argument as a **Function** (**Represents a function that accepts one argument and produces a result**)

- **Stream sorted():**
  We can use sorted() to sort the stream elements by passing Comparator argument.

Java 8 Stream API Limitations:

- **Stateless lambda expressions**:
  If you are using parallel stream and lambda expressions are stateful, it can result in random responses. The results are different because it depends on the way stream is getting iterated and we don't have any order defined for parallel processing. If we use sequential stream, then this problem will not arise.

- Once a Stream is consumed, it can't be used later on.

- There are a lot of methods in Stream API and the most confusing part is the overloaded methods. It makes the learning curve time taking.