Java 8 Functional Programing (Lamda Expresions, Functional Interface and Streams)

Lambda Expressions:

Q) What Lambda Calculus in Mathematics?

- To simply our calculation in mathematical expressions. To compute the big function expressions into small function expressions we can use lambda expressions
- Simplification with calculations
- Function/Expression Evaluation – with value/functionality – f(x) = x2+2x+2
- Anonymous (reference you cannot use in different place of another f(x,y,z) functions.
- Reference link : https://imgur.com/a/XBHub

Q) How many ways we can make use of Lambda expressions in java?

a. We know that lambda expression we can use implementations for functional interface abstract methods, it should match with method parameters and method return type
   FunctionalInterface reference = () -> { };
b. And also we can use in return statement which will have return types as functional interface.
   public Consumer<UserData> getConsumerInstance(p1,p2) { return userData -> {

```
                }
        }
```

- More generally, the types of parameters in a lambda expression must match the types of parameters of the single abstract method of a functional interface type, and the return types must match also.

Q) Scope of Lambda expressions when we compare with anonymous inner class? can we replace lambda expressions in all the places of anonymous inner classes?

- Lambda expressions can use variables defined in an **outer scope**. We refer to these lambdas as Capturing Lambdas.
- They can capture **static variables, instance variables, and local variables**, but only local variables must be **final or effectively final** (variable that never get changes after its initialization).

Following code will work fine-

```
int outer = 3;
Foo foo = () ->{
int inner = 4; //completely new variable local to lambda
inner++; //can modify inner
System.out.println(outer); //can access outer
System.out.println(inner); //can access inner
}
```

But this will not work-

```
int outer = 3;
int inner = 4;
Foo foo = () ->{
int inner = 4; //won't compile, trying to redeclare inner
System.out.println(outer);
System.out.println(inner);
}
```

```
int outer = 3;
outer+=1; //outer changed, no more effectively final
Foo foo = () ->{
System.out.println(outer); /*won't compile, cannot access outer as it is not final or effectively final*/

}
```

- First reason behind this is lambda is making a copy of outer (capturing outer) so forcing the variable to be final or effectively final avoids giving the impression that changing outer inside the lambda could actually modify the outer method parameter.
- But why lambda is making copy of outer ? Well because we can return the lambda from our method and as **soon as we return from this method its local variables get garbage collected** so Java has to make **a copy of local variables inside lambda in order for this lambda to live outside of this method**.

Second reason is concurrency issues, see below example.:

```
public void multithread() {
  boolean flag = true;
  executor.execute(() -> {
     while (flag) {
        // do something
   }
  });
  flag = false;  //flag updated, no longer effectively final
 }
```

- As each thread has its own stack, so how would while loop know that flag has been flipped to false in another thread.

- No restriction on static and instance variables to be final or effectively final because local variables are stored in stack while static and instance variables are stored in heap memory and lambda will have access to latest values of static or instance variables.

Reference link: https://medium.com/@lavishj77/java-lambda-expressions-4ea3b8245196#:~:text=Lambda%20expressions%20can%20use%20variables,get%20changes%20after%20its%20initialization).

Functional Interfaces:

- To make use lambda expressions in java we need functional interface only.
- Generally, interfaces can be used for references to on fly implementations or original implementations.
- Lambda expression is on fly implementation or function, so lambda expressions cast to interfaces. As Lambda expressions are anonymous we cannot mention the name of the interface method to make use.
- So Java people forcing us to keep only one method in interface to be automatically call the abstract method. This single abstract method interfaces are called Functional Interface.

Examples:

```
interface Sayable{
   public String say();
}
public class LambdaExpressionExample3{
public static void main(String[] args) {
   Sayable s=()->{
      return "I have nothing to say.";
   };
   System.out.println(s.say());
}
}
```

We do have some predefined functional interfaces provided java people along with java 8 version release and we do have before 1.7 version also but Functional interface topic came in java 1.8 only.

Java7 interfaces:

1. **Runnable** –

```
@FunctionalInterface
public interface Runnable {
   void run();
}
```

2. **Comparable**<T> :

```
public interface Comparable<T> {
   public int compareTo(T o);
}
```

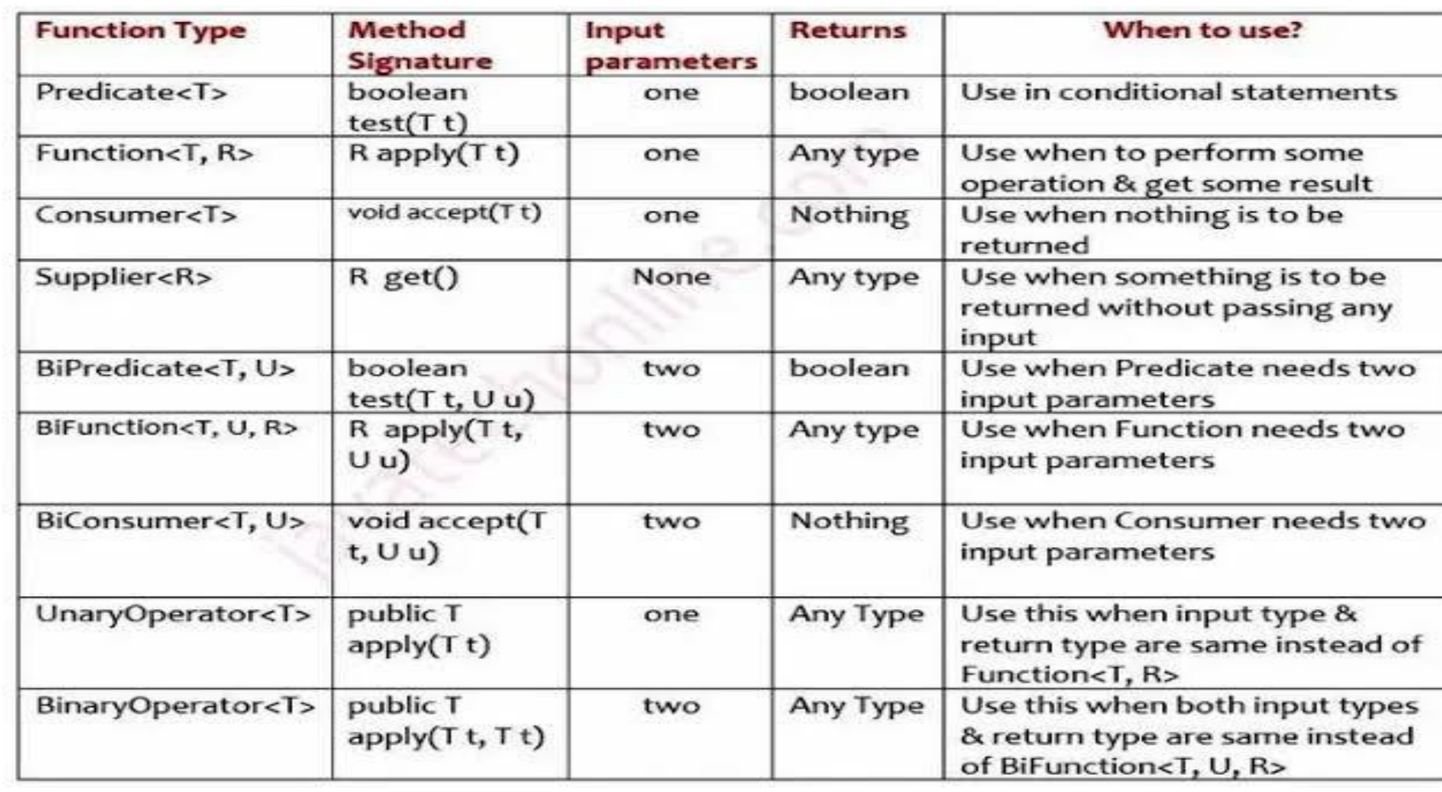
3. Comparator<T>:

```
@FunctionalInterface
public interface Comparator<T> {

   int compare(T o1, T o2);
}
```

4. Callable<V>:

```
@FunctionalInterface
public interface Callable<V> {

   V call() throws Exception;
}
```

<u>Java 8 predefined functional interfaces:</u>



Function Type	Method Signature	Input parameters	Returns	When to use?
Predicate<T>	boolean test(T t)	one	boolean	Use in conditional statements
Function<T, R>	R apply(T t)	one	Any type	Use when to perform some operation & get some result
Consumer<T>	void accept(T t)	one	Nothing	Use when nothing is to be returned
Supplier<R>	R get()	None	Any type	Use when something is to be returned without passing any input
BiPredicate<T, U>	boolean test(T t, U u)	two	boolean	Use when Predicate needs two input parameters
BiFunction<T, U, R>	R apply(T t, U u)	two	Any type	Use when Function needs two input parameters
BiConsumer<T, U>	void accept(T t, U u)	two	Nothing	Use when Consumer needs two input parameters
UnaryOperator<T>	public T apply(T t)	one	Any Type	Use this when input type & return type are same instead of Function<T, R>
BinaryOperator<T>	public T apply(T t, T t)	two	Any Type	Use this when both input types & return type are same instead of BiFunction<T, U, R>

Q) Why static and default method introduced in functional interfaces?

- They allow us to add new methods to an interface that are automatically available in the implementations. Therefore, we don't need to modify the implementing classes. In this way, **backward compatibility** is neatly preserved without having to refactor the implementers.
- Like regular interface methods, default methods are implicitly public; there's no need to specify the public modifier
- Unlike regular interface methods, we declare them with the default keyword at the beginning of the method signature, and they provide an implementation.
- The reason why the Java 8 release included default methods is pretty obvious.
- In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down

```
public interface Vehicle {

   String getBrand();

   String speedUp();

   String slowDown();

   default String turnAlarmOn() {
      return "Turning the vehicle alarm on.";
   }

   default String turnAlarmOff() {
      return "Turning the vehicle alarm off.";
   }
}
```

- In addition to declaring default methods in interfaces, Java 8 also allows us to define and implement static methods in interfaces.
- Since static methods don't belong to a particular object, they're not part of the API of the classes implementing the interface; therefore, they have to be called by using the interface name preceding the method name.

```
public interface Vehicle {

   // regular / default interface methods

   static int getHorsePower(int rpm, int torque) {
      return (rpm * torque) / 5252;
   }
}
```

- Defining a static method within an interface is identical to defining one in a class. Moreover, a static method can be invoked within other static and default methods.
- The idea behind static interface methods is to provide a simple mechanism that allows us to increase the degree of cohesion of a design by putting together related methods in one single place without having to create an object.
- The same can pretty much be done with abstract classes. The main difference is that abstract classes can have constructors, state, and behavior.

Q) How did java overcome ambiguity nature when we have same parameters methods/return type methods in both functional interface and those two interface implemented by one class?

- **We must override in child classes at the compile time.**

Example:

```
Interface I1 {
default int m1(int i) {
return 1;
}
}

Interface I1 {
default int m1(int i) {
return 1;
}
}
```

```
class Main implements I1, I2 {
    p s v m() {
           Main m =new Mian();
           m.I1.m1();//CTE---- here compiler will force you to override m1 methods as it has ambiguity to
call which m1 methods as it has two parents
   }
}
```

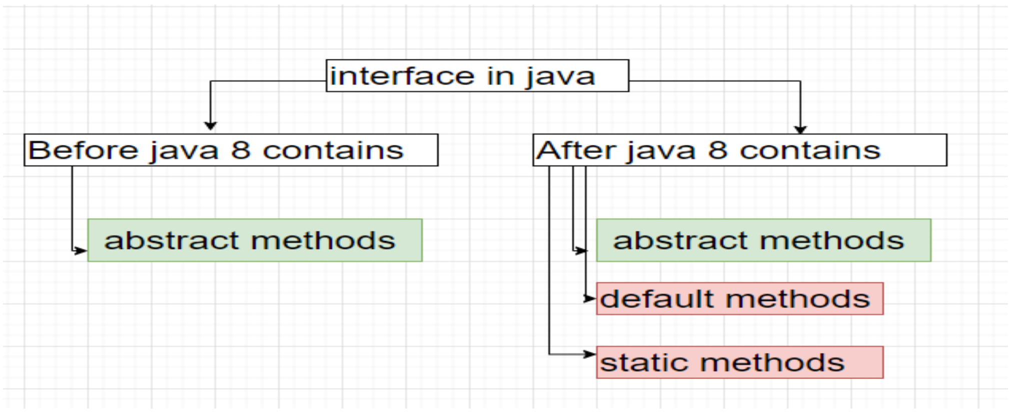
Difference between static and default methods in java8 interface:

Why static methods are introduced in java1.8:

- Have you ever created utility static classes which holds utility methods ? Remember Collections class which actually has many static method which are actually utility methods for Collection interface.
- Isn't it good if you able to create utility methods in the interface itself It 'll save creating additional class. Using static methods in interfaces can be compared to write static utility methods for a class in separate class.

- In the past, if you had an interface Foo and wanted to group interface-related utils or factory methods, you would need to create a separate utils class FooUtils and store everything there.
- Those classes would not have anything in common other than the name, and additionally, the utils class would need to be made final and have a private constructor to forbid unwanted usage.
- Now, thanks to the interface static methods, you can keep everything in one place without creating any additional classes.
- It's also important to not forget all good practices and not throw everything mindlessly to one interface class

- 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 incase of poor implementation in implementation classes.



- Interfaces are actually real drivers behind abstraction and polymorphism in java.
- But what if we have one interface which is implemented in 100 classes and now we want to introduce new method in the interface?

Q) Why we need default and static methods in java?

- The real benefit for default method in interface is for API and libraries developer. If any new method is introduced in the interface if we use default method, it’s not needed to change all implemented classes as all classes will inherit them automatically.
- Static methods we can use like utility methods and we can call only through interface name.
- We can’t override static methods as it’s part of class/interface loading time it will get loaded in to JVM memory’.

<u>There are two main difference between Abstract Class and Interface in java 8.</u>

1. Abstract classes can have instance variables however interface cannot

2. Class can extend only from one abstract class, but a class can implement multiple interfaces.

Method References:

Where can you use the double colon operator in Java?

You can use the double colon operator (::) wherever you need to use the method reference. Here are some examples of a method reference in Java 8:

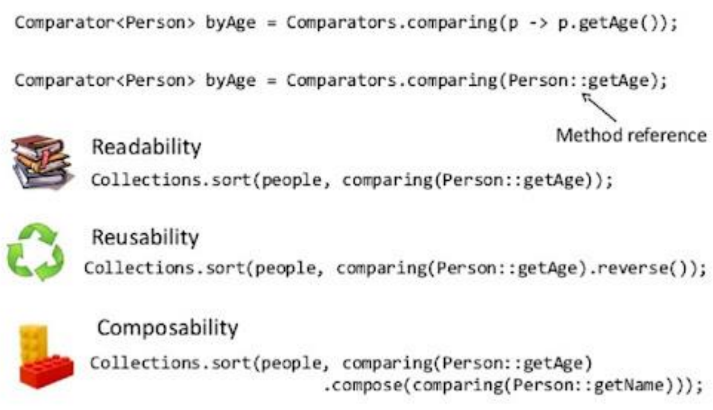
- A static method (ClassName::methodName) like Person::getAge
- An instance method of a particular object (instanceRef::methodName) like System.out::println
- A super method of a particular object (super::methodName)
- An instance method of an arbitrary object of a particular type (ClassName::methodName)—existing object type
- A class constructor reference (ClassName::new) like ArrayList::new
- An array constructor reference (TypeName[]::new) like String[]::new

<u>Reference to an Instance Method of an Arbitrary Object of a Particular Type</u>

The following is an example of a reference to an instance method of an arbitrary object of a particular type:

```
String[] stringArray = { "Barbara", "James", "Mary", "John",
   "Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

- The equivalent lambda expression for the method reference String::compareToIgnoreCase would have the formal parameter list (String a, String b), where a and b are arbitrary names used to better describe this example. The method reference would invoke the method a.compareToIgnoreCase(b).
- Similarly, the method reference String::concat would invoke the method a.concat(b).



```
Comparator<Person> byAge = Comparators.comparing(p -> p.getAge());

Comparator<Person> byAge = Comparators.comparing(Person::getAge);

Collections.sort(people, comparing(Person::getAge));

Collections.sort(people, comparing(Person::getAge).reverse());

Collections.sort(people, comparing(Person::getAge)
                          .compose(comparing(Person::getName)));
```

Q) How method references double colon operator is working in java internally?

Method reference is used to refer abstract method of a functional interface with a concrete method of another class. The class that uses method reference doesn't need to implement the interface. It is compact form of lambda expression.

Q) Method references priority for same name variable in java?

If the first search produces a static method, and no non-static method is applicable [..], then the compile-time declaration is the result of the first search. Otherwise, if no static method is applicable [..], and the second search produces a non-static method, then the compile-time declaration is the result of the second search. Otherwise, there is no compile-time declaration.

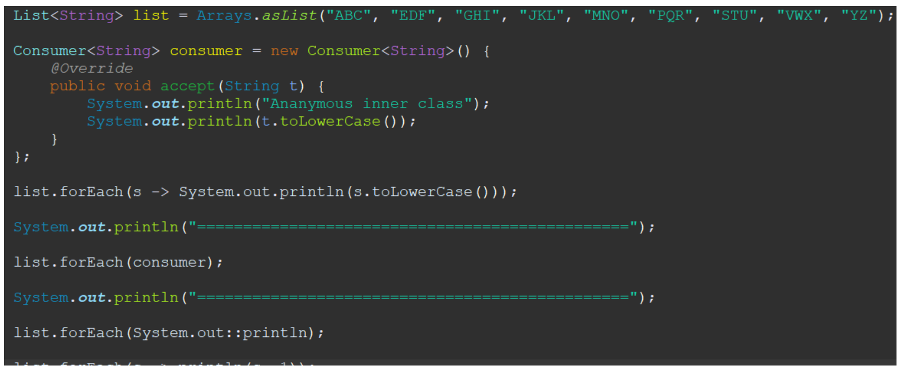
```
List<A> a = Arrays.asList(new A(2), new A(3));
                        a.stream().filter(b -> b.is()).forEach(System.out::println);;
                        a.stream().filter(A::is);
```

Reference Link : https://stackoverflow.com/questions/23051879/java-8-reference-to-a-static-method-vs-instance-method

<u>Is Method References being replacements of LamdaExpresions:</u>

Answer is 'No' – lamada expressions we can call existing methods and write logical statements also.

But we will use Method References, we can use



```
List<String> list = Arrays.asList("ABC", "EDF", "GHI", "JKL", "MNO", "PQR", "STU", "VWX", "YZ");

Consumer<String> consumer = new Consumer<String>() {
    @Override
    public void accept(String t) {
        System.out.println("Ananymous inner class");
        System.out.println(t.toLowerCase());
    }
};

list.forEach(s -> System.out.println(s.toLowerCase()));

System.out.println("===============================================");

list.forEach(consumer);

System.out.println("===============================================");

list.forEach(System.out::println);
```

<u>Introduction to Method Reference</u>

- **object::methodName**

Let's create class MethodReferecesLambdaExpressionMain in which we are going to print list using stream's foreach method.

```
public class MethodReferecesLambdaExpressionMain {

   public static void main(String args[])
   {
      List<String> countryList=Arrays.asList(new String[] {"India", "China","Nepal","Russia"});

      System.out.println("======================");
      System.out.println("Using anonymous class");
```

```
     System.out.println("======================");

     // Using anonymous class
     countryList.stream().forEach(
           new Consumer<String>() {

              @Override
              public void accept(String country) {
                 System.out.println(country);
              }
           });


     System.out.println("======================");
     System.out.println("Using lambda expression");
     System.out.println("======================");

   // Using lambda expression
     countryList.stream().forEach(
           country -> System.out.println(country)
           );

     System.out.println("======================");
     System.out.println("Using Method references");
     System.out.println("======================");

    // Using method reference
     countryList.stream().forEach(
           System.out::println
        );
   }
}
```

Output:

```
======================
Using anonymous class
======================
India
China
Nepal
Russia
======================
Using lambda expression
======================
India
China
Nepal
Russia
```

```
=======================
Using Method references
=======================
India
China
Nepal
Russia
```

Note: stream.foreach() method takes consumer functional interface as agrument.

Consumer is functional interface that takes a single argument and returns nothing.

We have used consumer functional interface in 3 ways.

1. **Using anonymous class:**

```
  Consumer<string> consumer1 = new Consumer<string>() {

            @Override
            public void accept(String country) {
              System.out.println(country);
            }
        };
```

2. **Using lambda expression**

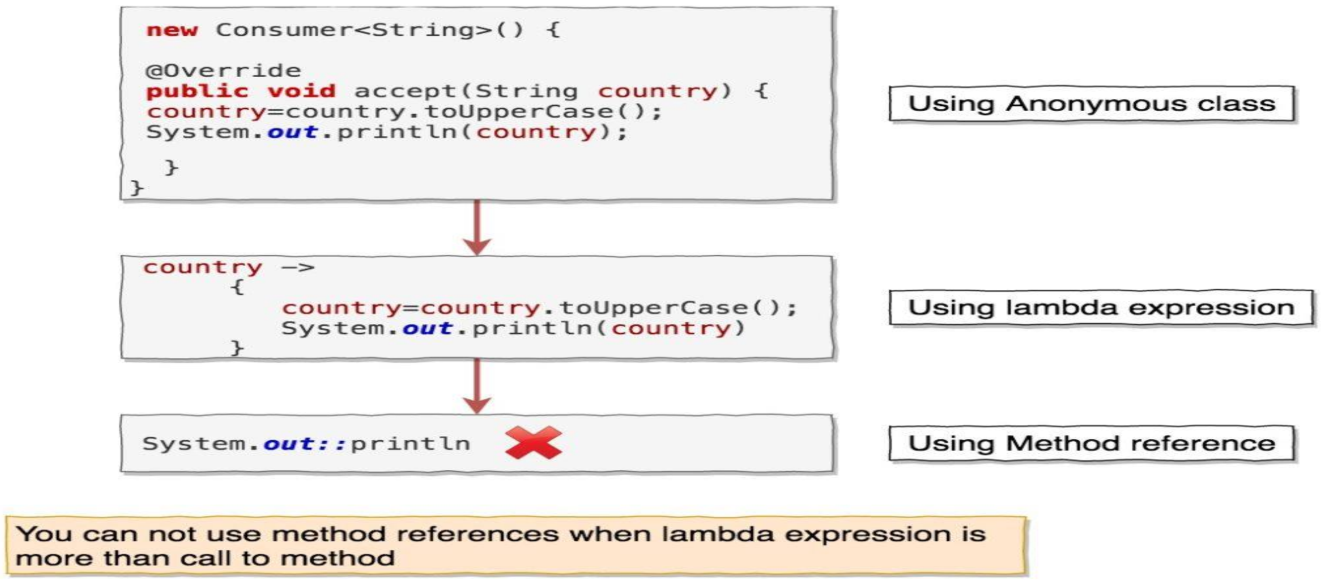
```
Consumer<string> consumer2 = country -> System.out.println(country);
```

3. **Using method reference**

```
Consumer<string> consumer3 = System.out::println;
```

- Important note: You might already know that you can use lambda expression instead of an anonymous class, but You can use method reference only when the lambda expression just calls to a method.
- In method reference, we have Class or object before :: and method name after :: without arguments.
- Did you notice the method reference does not have arguments? Yes, we don't need to pass arguments to method reference, arguments are passed automatically internally based on type of method reference.

Let's say you want to convert country to uppercase before printing it. You can achieve it using anonymous class and lambda expression but not with method reference.



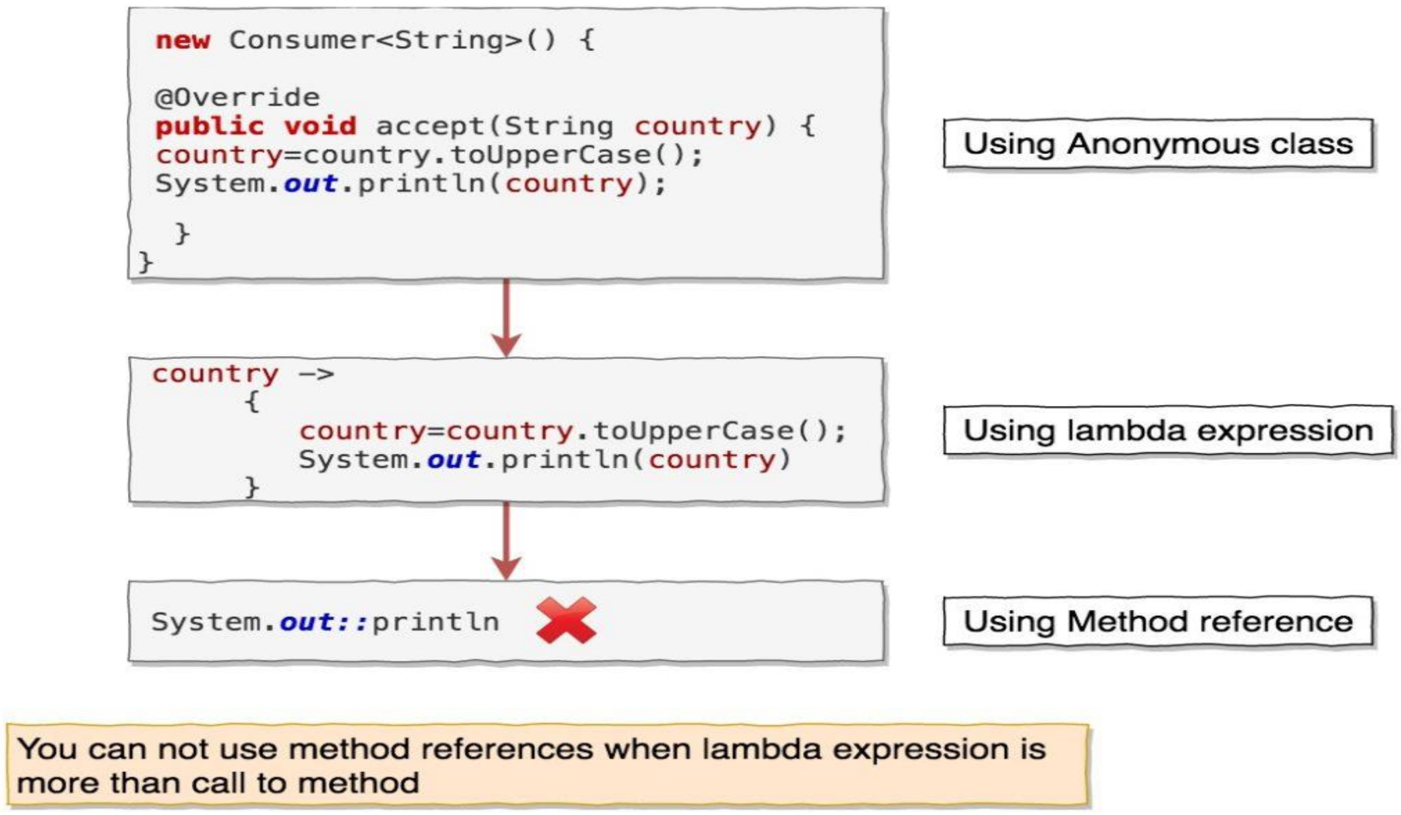
```
new Consumer<String>() {

@Override
public void accept(String country) {
country=country.toUpperCase();
System.out.println(country);

  }
}
```

```
country ->
      {
         country=country.toUpperCase();
         System.out.println(country)
      }
```

```
System.out::println
```

You can not use method references when lambda expression is more than call to method

You cannot use method reference as below:



```
new Consumer<String>() {

@Override
public void accept(String country) {
country=country.toUpperCase();
System.out.println(country);

  }
}
```

```
country ->
      {
         country=country.toUpperCase();
         System.out.println(country)
      }
```

```
System.out::println
```

You can not use method references when lambda expression is more than call to method

<u>Types of method references:</u>

There are four types of method references.

- ✓ Reference to static method
- ✓ Reference to instance method of object type

- ✓ Reference to instance method of existing object
- ✓ Reference constructor

1. Reference to static method:

When you have lambda expression which calls to static method, then you can method reference to static method.

Lambda expression syntax

(args) -> ClassName.someStaticMethod(args)

can be converted to

ClassName::someStaticMethod

Let’s see this with the help of example. Create a class name PowerFunctions

```
class PowerFunctions {

  // This is the method we will call in method reference
  public static Integer power(int a)
  {
     return a*a;
  }

  // Function is functional interface which will be target for method reference
  public static List<Integer> calculatePowOf2ForList(List<Integer> list,
                               Function<Integer,Integer> function)
  {
     List<Integer> powerNumbers = new ArrayList<>();

     for(Integer num:list)
     {

        Integer powOf2 = function.apply(num);
        powerNumbers.add(powOf2);
     }
     return powerNumbers;
  }

}
```

Function is functional interface that takes a single input T and returns a single output R. We can call calculatePowOf2ForList() as below:

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
```

```
public class StaticMethodReferenceMain {

    public static void main(String args[])
    {

        List<Integer> list=Arrays.asList(new Integer[] {1,2,3,4,5});

        // using anonymous class
        Function<Integer,Integer> function1=new Function<Integer, Integer>() {

            @Override
            public Integer apply(Integer num) {
                return PowerFunctions.power(num);
            }
        };

        List<Integer> calculatePowForList1 = PowerFunctions.calculatePowOf2ForList(list, function1);
        System.out.println(calculatePowForList1);


        // Using lambda expression
        Function<Integer,Integer> function2 = (num) -> PowerFunctions.power(num);

        List<Integer> calculatePowForList2 = PowerFunctions.calculatePowOf2ForList(list, function2);
        System.out.println(calculatePowForList2);

        // Using Method reference
        Function<Integer,Integer> function3 = PowerFunctions::power;

        List<Integer> calculatePowForList3 = PowerFunctions.calculatePowOf2ForList(list, function3);
        System.out.println(calculatePowForList3);

    }
}
```

When you run above program, you will get below output:
Output

```
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
```

If you notice,Function<Integer,Integer> function2 = (num) -> PowerFunctions.power(num); is of type (args) -> className.someStaticMethod(args)

Here,

PowerFunctions is className

someStaticMethod is power method

num is power method argument.

We are calling a static method power of class PowerFunctions in lambda expression, that's why we can use it as method reference. So instead of

Function<Integer,Integer> function2 = (num) -> PowerFunctions.power(num);

we can use

Function<Integer,Integer> function3 = PowerFunctions::power;

Here,

First type parameter of Function(Integer) is first parameter of static method power().

Second type parameter of Function(Integer) is return type of static method power().

2. **Reference to instance method of object type :**

When you have lambda expression where instance of object is passed and calls to an instance method with/without parameters, then you can use method reference to an instance method with object type.

Lambda expression syntax

(obj,args) -> obj.someInstanceMethod(args)

can be converted to

objectType::someInstanceMethod

Let's see this with the help of example.

```
import java.util.function.BiFunction;

public class MethodReferenceObjectType {

    public static void main(String[] args) {

        // Using anonymous class


        BiFunction<String,Integer,String> bf1=new BiFunction<>() {

            @Override
            public String apply(String t, Integer u) {
                return t.substring(u);
            }
        };
        String subString1 = getSubstring("Java2blog",2,bf1);
        System.out.println(subString1);
        // Using lambda expression
        BiFunction<String,Integer,String> bf2 =  (t,u) -> t.substring(u);
        String subString2 = getSubstring("Java2blog",2,bf2);
        System.out.println(subString2);

        // Using Method reference
        BiFunction<String,Integer,String> bf3 = String::substring;
        String subString3 = getSubstring("Java2blog",2,bf3);
        System.out.println(subString3);
    }


    public static String getSubstring(String str1,int beginIndex,BiFunction<String,Integer,String> p)
    {
        return p.apply(str1, beginIndex);

    }
}
```

BiFunction is functional interface that takes two arguments and returns single output.

If you notice,BiFunction<String,Integer,String> bf2 = (t,u) -> t.substring(u); is of type (obj,args) -> obj.someInstanceMethod(args)

Here

obj is of type String.

someInstanceMethod is String's substring() method.

args is beginIndex for substring() method argument.

So BiFunction<String,Integer,String> bf2 = (t,u) -> t.substring(u); can be converted to

BiFunction<String,Integer,String> bf3 = String::substring;

Here,

First BiFunction parameter type(String) is String object itself.

Second BiFunction parameter type(Integer) is argument to substring() method

Third BiFunction parameter type(String) is return type of substring() method

3. **Reference to instance method of existing object:**

Important point:

When you pass list of Strings/List of Person user defined data types, then we can apply arbitrary object reference with type of the class for any instance methods. For example

```
List<String> list = Ararys.asList("ABC", "DEF");
List.stream().map(Person::toLowerCase).collect(Collectors.toList());
```

Explanation: it will be applicable when you pass list of objects and that object reference already holding by that loop indexes, for above example list.stream()→ you will get input as string object so you no need to create separate string object, by type of that object you can invoke all the instance methods:

When you have lambda expression where instance of object is used to call an instance method with/without parameters, then you can use method reference to an instance method with an existing object.

Lambda expression syntax

(args) -> obj.someInstanceMethod(args)

can be converted to

objectType::someInstanceMethod

Here obj is defined somewhere else and is not part of argument to lambda expression.Let's understand with the help of example. Create a class named Country.java.

```
public class Country {
    String name;
    long population;

    Country(String name)
    {
        this.name=name;
    }
    public String getName() {
        return name;

    }
    public void setName(String name) {
        this.name = name;
    }
    public long getPopulation() {
        return population;
    }
    public void setPopulation(long population) {
        this.population = population;
    }

    @Override
    public String toString() {
        return "[ name = "+name+" population = "+population+" ]";
    }
}
```

Create another class MethodReferenceExistingObjectMain.java

```
import java.util.function.Consumer;

public class MethodReferenceExistingObjectMain {

    public static void main(String[] args) {

        Country c=new Country("India");

        // Using anonymous class
        Consumer<Long> popCons1=new Consumer<Long>() {

            @Override
            public void accept(Long t) {
                c.setPopulation(t);
            }
        };
        popCons1.accept(20000L);
        System.out.println(c);


        // Using Lambda expression
        Consumer<Long> popCons2= (population) -> c.setPopulation(population);
        popCons2.accept(30000L);
        System.out.println(c);

        // Using method reference
        Consumer<Long> popCons3 = c::setPopulation;
        popCons3.accept(40000L);
        System.out.println(c);
    }
}
```

Output

```
[ name = India population = 20000 ]
[ name = India population = 30000 ]
[ name = India population = 4000
```

Consumer is functional interface which takes single argument and returns nothing.

If you notice, Consumer popCons2 = (population) -> c.setPopulation(population); is of type (args) -> obj.someInstanceMethod(args) Here

obj is of type Country and declared somewhere else.

someInstanceMethod is Country’s setPopulation method.

args is population for setPopulation method argument.

So Consumer<Long> popCons2= (population) -> c.setPopulation(population); can be converted to Consumer<Long> popCons3 = c::setPopulation;

Here, First Consumer parameter type(Long) is argument to setPopulation method.

4.Reference constructor

When lambda expression is used to create new object with/without parameters, then you can use reference method constructor.

Lambda expression syntax

(args) -> new ClassName(args)

can be converted to

ClassName::new

Let's see with the help of example. Here we will convert the list to set using method reference. Function<List,Set> is functional interface which will take a list an argument and will return set by calling HashSet constructor public HashSet(Collection<? extends E> c)

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.function.Function;

public class MethodReferenceConstructorMain {

    public static void main(String[] args) {

        ArrayList<String> list=new ArrayList<>();
        list.add("Rohan");
        list.add("Andy");
        list.add("Sneha");
        list.add("Rohan");

        // Anonymous class
        Function<List<String>,Set<String>> f1= new Function<List<String>, Set<String>>() {


            @Override
            public Set<String> apply(List<String> nameList) {

                return new HashSet<>(nameList);
            }
        };
        Set<String> set1 = f1.apply(list);
        System.out.println(set1);
        // Using lambda expression
        Function<List<String>,Set<String>> f2 = (nameList) -> new HashSet<>(nameList);
        Set<String> set2 = f2.apply(list);
        System.out.println(set2);

        // Using Method reference
        Function<List<String>,Set<String>> f3= HashSet::new;
        Set<String> set = f3.apply(list);
        System.out.println(set);
    }
}

Output

[Sneha, Andy, Rohan]
[Sneha, Andy, Rohan]
[Sneha, Andy, Rohan]
```

If you notice, Function<List<String>,Set<String>> f2 = (nameList) -> new HashSet<>(nameList); is of type (args) -> new ClassName(args)

Here

args is of type list

ClassName is HashSet

So Function<List<String>,Set<String>> f2 = (nameList) -> new HashSet<>(nameList); can be converted to Function<List<String>,Set<String>> f3= HashSet::new;

Here,

First Function parameter type(List) is argument to HashSet constructor.

Reference link : https://java2blog.com/java-8-method-reference/

Streams:

Q) Can we do whatever stream will do the functionality using collections, if yes then why do we need to go for streams and what is difference between Streams and Collections?

1. Major difference is collections and its methods **operates/work on source/data structure** directly that means it will **modify/change the structure** of source(list,set,etc).
2. **Streams** never modify the source data, it will process the data and it has its own operations like internal methods and terminal methods which will produce another collections object based on our requirements.

3. Streams operations are immutable.

There are many ways to create a stream instance of different sources. Once created, the instance will not modify its source, therefore allowing the creation of multiple instances from a single source.

<u>How many ways we can create Stream object?</u>

<u>1. Stream of Array:</u>

- An array can also be the source of a stream:

```
Stream<String> streamOfArray = Stream.of("a", "b", "c");
```

- We can also create a stream out of an existing array or of part of an array:

```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> streamOfArrayFull = Arrays.stream(arr);
Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

- We can also create a stream of any type of Collection **(Collection, List, Set)**

```
Collection<String> collection = Arrays.asList("a", "b", "c");
Stream<String> streamOfCollection = collection.stream();
```

<u>Note</u>: We don’t have direct method to create **Set/Map** from existing java classes, we can use third party libraries for that.

<u>2.Stream.iterate():</u>

- Another way of creating an infinite stream is by using the iterate() method:

```
Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);
```

- The first element of the resulting stream is the first parameter of the iterate() method. When creating every following element, the specified function is applied to the previous element. In the example above the second element will be 42.

<u>3.Stream of String:</u>

✓ We can also use String as a source for creating a stream with the help of the **chars**() method of the String class.

✓ Since there is no interface for CharStream in JDK, we use the IntStream to represent a stream of chars instead.

```
IntStream streamOfChars = "abc".chars();
```

✓ The following example breaks a String into sub-strings according to specified RegEx:

```
Stream<String> streamOfString =  Pattern.compile(", ").splitAsStream("a, b, c");
```

<u>4. Stream of File:</u>

✓ Furthermore, Java NIO class Files allows us to generate a Stream<String> of a text file through the **lines**() method.

✓ Every line of the text becomes an element of the stream:

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings = Files.lines(path);
Stream<String> streamWithCharset =
  Files.lines(path, Charset.forName("UTF-8"));
```

The Charset can be specified as an argument of the lines() method.

5. Referencing a Stream:

Important point:

- ✓ We can instantiate a stream, and have an accessible reference to it, as long as only intermediate operations are called.
- ✓ After executing a terminal operation makes a stream **inaccessible**.

Example:

- ✓ To demonstrate this, we will forget for a while that the best practice is to chain the sequence of operation.
- ✓ Besides its unnecessary verbosity, technically the following code is valid:

```
Stream<String> stream =
  Stream.of("a", "b", "c").filter(element -> element.contains("b"));
Optional<String> anyElement = stream.findAny();
```

- ✓ However, an attempt to reuse the same reference after calling the terminal operation will trigger the IllegalStateException: This kind of behavior is logical. We designed streams to apply a finite sequence of operations to the source of elements in a functional style, not to store elements.

```
Optional<String> firstElement = stream.findFirst();
```

As the IllegalStateException is a RuntimeException, a compiler will not signalize about a problem. So it is very important to remember that Java 8 streams can't be reused.

So to make the previous code work properly, some changes should be made:

```
List<String> elements =  Stream.of("a", "b", "c").filter(element -> element.contains("b"))
.collect(Collectors.toList());
Optional<String> anyElement = elements.stream().findAny();
Optional<String> firstElement = elements.stream().findFirst();
```

6. Stream Pipeline:

- ✓ To perform a sequence of operations over the elements of the data source and aggregate their results, we need three parts: **the source, intermediate operation(s) and a terminal operation.**
- ✓ Intermediate operations return a new modified stream. For example, to create a new stream of the existing one without few elements, the skip() method should be used:

```
Stream<String> onceModifiedStream =  Stream.of("abcd", "bbcd", "cbcd").skip(1);
```

- ✓ If we need more than one modification, we can chain intermediate operations. Let's assume that we also need to substitute every element of the current Stream<String> with a **sub-string of** the first few chars. We can do this by chaining the **skip**() and **map**() methods:

```
Stream<String> twiceModifiedStream =
stream.skip(1).map(element -> element.substring(0, 3));
```

- ✓ A stream by itself is worthless; the user is interested in the result of the terminal operation, which can be a value of some type or an action applied to every element of the stream. We can only use one terminal operation per stream.
- ✓ The correct and most convenient way to use streams is by a stream pipeline, which is a chain of the stream source, intermediate operations, and a terminal operation:

```
List<String> list = Arrays.asList("abc1", "abc2", "abc3");
long size = list.stream().skip(1)
  .map(element -> element.substring(0, 3)).sorted().count();
```

7. Lazy Invocation:

- ✓ Intermediate operations are lazy. This means that they will be invoked only if it is necessary for the terminal operation execution.
- ✓ For example, let's call the method wasCalled(), which increments an inner counter every time it's called:

Example:

```
private long counter;//Instance attribute

private void wasCalled() {
   counter++;
}
```

- ✓ Now let's call the method wasCalled() from operation filter():

```
List<String> list = Arrays.asList("abc1", "abc2", "abc3");
counter = 0;
Stream<String> stream = list.stream().filter(element -> {
   wasCalled();
   return element.contains("2");
});
```

- ➢ As we have a source of three elements, we can assume that the filter() method will be called three times, and the value of the counter variable will be 3.
- ➢ However, running this code doesn't change counter at all, it is still zero, so the filter() method wasn't even called once.
- ➢ The reason why is missing of the terminal operation.

Let’s rewrite this code a little bit by adding a map() operation and a terminal operation, findFirst(). We will also add the ability to track the order of method calls with the help of logging

```
Optional<String> stream = list.stream().filter(element -> {
    log.info("filter() was called");
    return element.contains("2");
}).map(element -> {
    log.info("map() was called");
    return element.toUpperCase();
}).findFirst();
```

- The resulting log shows that we called the filter() method twice and the map() method once.
- This is because the pipeline executes vertically. In our example, the first element of the stream didn’t satisfy the filter’s predicate.
- Then we invoked the filter() method for the second element, which passed the filter. Without calling the filter() for the third element, we went down through the pipeline to the map() method.
- The findFirst() operation satisfies by just one element. So in this particular example, the lazy invocation allowed us to avoid two method calls, one for the filter() and one for the map().

8. Order of Execution:

- From the performance point of view, the right order is one of the most important aspects of chaining operations in the stream pipeline:

```
long size = list.stream().map(element -> {
    wasCalled();
    return element.substring(0, 3);
}).skip(2).count();
```

- Execution of this code will increase the value of the counter by three. This means that we called the map() method of the stream three times, but the value of the size is one.
- So the resulting stream has just one element, and we executed the expensive map() operations for no reason two out of the three times.

If we change the order of the skip() and the map() methods, the counter will increase by only one. So we will call the map() method only once:

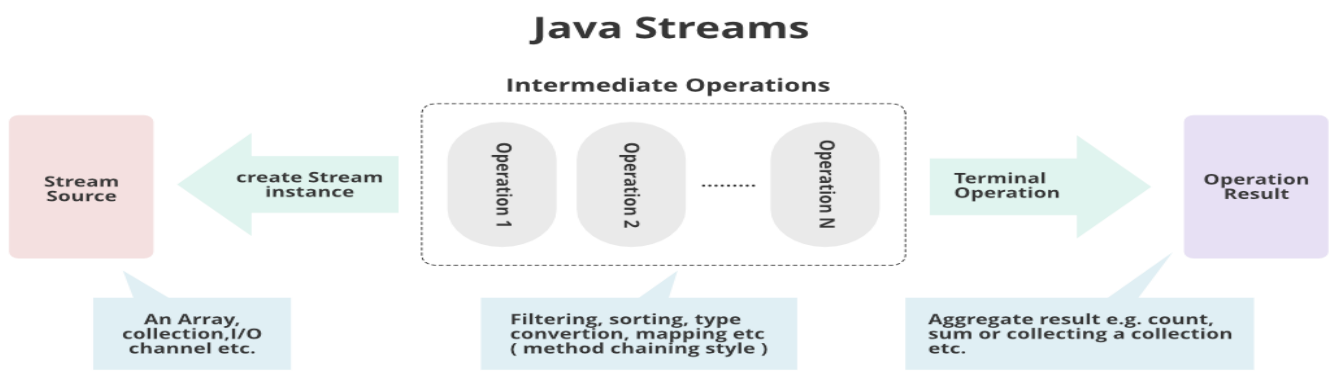
```
long size = list.stream().skip(2).map(element -> {
    wasCalled();
    return element.substring(0, 3);
}).count();
```

- This brings us to the following rule: intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element.

- So we need to keep methods such as skip(), filter(), and distinct() at the top of our stream pipeline.

9. Stream Reduction:

- The API has many terminal operations which aggregate a stream to a type or to a primitive: count(), max(), min(), and sum().
- However, these operations work according to the predefined implementation.
- So what if a developer needs to customize a Stream's reduction mechanism? There are two methods which allow us to do this,
  1. the **reduce**() and
  2. the **collect**() methods.


Java Streams
Intermediate Operations
Stream Source
create Stream instance
Operation 1
Operation 2
Operation N
Terminal Operation
Operation Result
An Array, collection,I/O channel etc.
Filtering, sorting, type convertion, mapping etc (method chaining style)
Aggregate result e.g. count, sum or collecting a collection etc.

The reduce() Method:

There are three variations of this method, which differ by their signatures and returning types. They can have the following parameters:

1. **identity** – the initial value for an accumulator, or a default value if a stream is empty and there is nothing to accumulate
2. **accumulator** – a function which specifies the logic of the aggregation of elements. As the accumulator creates a new value for every step of reducing, the quantity of new values equals the stream's size and only the last value is useful. This is not very good for the performance.
3. **combiner** – a function which aggregates the results of the accumulator. We only call combiner in a parallel mode to reduce the results of accumulators from different threads.

Now let's look at these three methods in action:

```
OptionalInt reduced =  IntStream.range(1, 4).reduce((a, b) -> a + b);
```
Output: reduced = 6 (1 + 2 + 3)

```
int reducedTwoParams =
  IntStream.range(1, 4).reduce(10, (a, b) -> a + b);
```
Output: reducedTwoParams = 16 (10 + 1 + 2 + 3)

```
int reducedParams = Stream.of(1, 2, 3)
  .reduce(10, (a, b) -> a + b, (a, b) -> {
     log.info("combiner was called");
     return a + b;
  });
```
Note: The result will be the same as in the previous example (16), and there will be no login, which means that combiner wasn't called.

To make a combiner work, a stream should be parallel:

```
int reducedParallel = Arrays.asList(1, 2, 3).parallelStream()
    .reduce(10, (a, b) -> a + b, (a, b) -> {
       log.info("combiner was called");
       return a + b;
    });
```
Explanation:

- The result here is different (36), and the combiner was called twice. Here the reduction works by the following algorithm: the accumulator ran three times by adding every element of the stream to identity. These actions are being done in parallel.
- As a result, they have (10 + 1 = 11; 10 + 2 = 12; 10 + 3 = 13;). Now combiner can merge these three results. It needs two iterations for that (12 + 13 = 25; 25 + 11 = 36).

The collect() Method:

- The reduction of a stream can also be executed by another terminal operation, the collect() method.
- It accepts an argument of the type Collector, which specifies the mechanism of reduction.
- There are already created, predefined collectors for most common operations.
- They can be accessed with the help of the Collectors type.

```
List<Product> productList = Arrays.asList(new Product(23, "potatoes"),
  new Product(14, "orange"), new Product(13, "lemon"),
  new Product(23, "bread"), new Product(13, "sugar"));
```

Converting a stream to the Collection (Collection, List or Set):

```
List<String> collectorCollection
productList.stream().map(Product::getName).collect(Collectors.toList());
```

Reducing to String:

```
String listToString = productList.stream().map(Product::getName)  .collect(Collectors.joining(", ", "[",
"]"));
```

Explanation:

- ✓ The joiner() method can have from one to three parameters (delimiter, prefix, suffix).
- ✓ The most convenient thing about using joiner() is that the developer doesn't need to check if the stream reaches its end to apply the suffix and not to apply a delimiter.
- ✓ Collector will take care of that.

Processing the average value of all numeric elements of the stream:

```
double averagePrice = productList.stream() .collect(Collectors.averagingInt(Product::getPrice));
```

Explanation:

The methods averagingXX(), summingXX() and summarizingXX() can work with primitives (int, long, double) and with their wrapper classes (Integer, Long, Double). One more powerful feature of these methods is providing the mapping. As a result, the developer doesn't need to use an additional map() operation before the collect() method.

10. Parallel Streams:

- ✓ Before Java 8, parallelization was complex. The emergence of the **ExecutorService and the ForkJoin simplified a developer's** life a little bit, but it was still worth remembering how to create a specific executor, how to run it, and so on. Java 8 introduced a way of accomplishing parallelism in a functional style.
- ✓ The API allows us to create parallel streams, which perform operations in a parallel mode. When the source of a stream is a Collection or an array, it can be achieved with the help of the parallelStream() method:

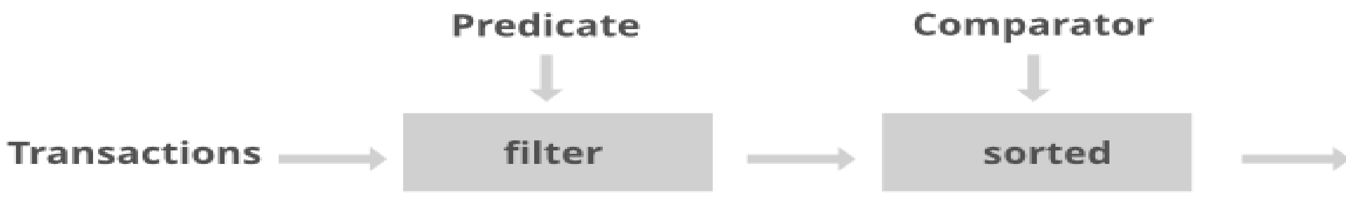
```
Stream<Product> streamOfCollection = productList.parallelStream();
boolean isParallel = streamOfCollection.isParallel();
boolean bigPrice = streamOfCollection
  .map(product -> product.getPrice() * 12)
  .anyMatch(price -> price > 200);
```

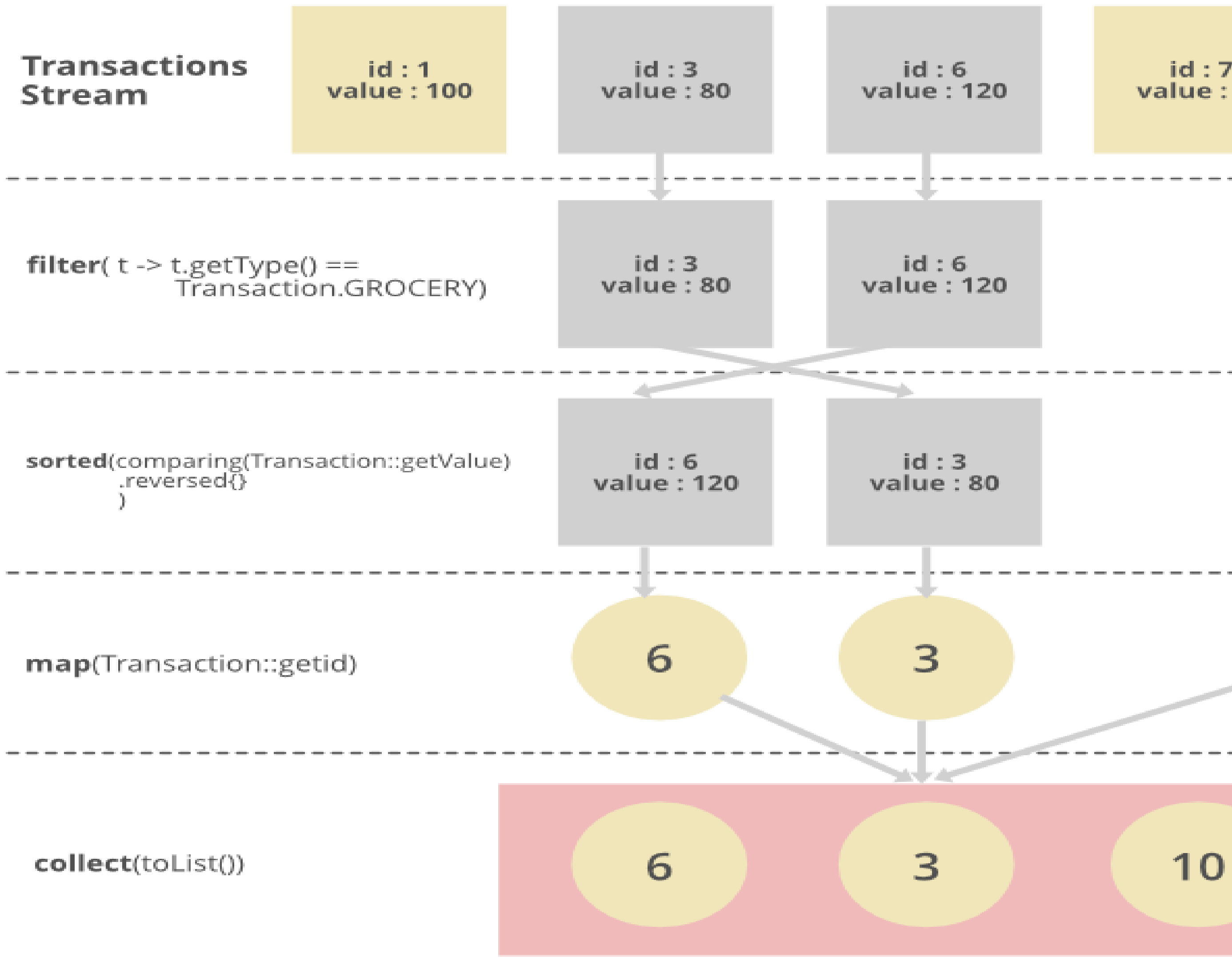
If the source of a stream is something other than a Collection or an array, the parallel() method should be used:

```
IntStream intStreamParallel = IntStream.range(1, 150).parallel();
boolean isParallel = intStreamParallel.isParallel();
```

Conclusion:

- The Stream API is a powerful, but simple to understand set of tools for processing the sequence of elements. When used properly, it allows us to reduce a huge amount of boilerplate code, create more readable programs, and improve an app's productivity.
- In most of the code samples shown in this article, we left the streams unconsumed (we didn't apply the close() method or a terminal operation). In a real app, don't leave an instantiated stream unconsumed, as that will lead to memory leaks.
- The complete code samples that accompany this article are available.





Stream Methods:

S No	Method name	Method description/situation	Input	Output	Example Program

1	**empty**()
	
	Stream<String> streamEmpty = Stream.empty();
		We should use the **empty()** method in case of the creation of an empty stream We often use the empty() method upon creation to	NA	Stream<T>	public Stream<String> streamC
					return list == null ||
					list.isEmpty() ?
					Stream.empty() : list.stream();
					}

		avoid returning null for streams with no element			
2	stream()	Returns a sequential stream considering collection as its source.			List<String> strings = Arrays.as "abcd","", "jkl"); List<String> filtered = strings.st !string.isEmpty()).collect(Collec
3	parallelStream()	Returns a parallel Stream considering collection as its source.			
4	of()				
5	filter()				
6	map()				
7	reduce()				
8	collect()				
9	concat()				
10	flatmap()				
11	skip()				
12	limit()				
13	count()				
14	Sorted()				
15	mapToInt()				
16	mapToObject()				
17	distinct()				
18	forEach				
19					
20					
21					

Collectors Methods:

Source for below collectors methods methods:

```
List<Employee> list = Arrays.asList(
                                new Employee(101, "Ramesh","Bonalu", "User", 10000, "Male",51),
                                new Employee(102, "Prabhu", "Gunapalu","Admin", 20000,"Male",
29),
                                new Employee(108, "Bala", "Thopu", "User", 10000, "Male",31),
                                new Employee(111, "Rupesh", "Reddy", "Hardware", 13000,
"Male",21),
                                new Employee(212, "Anirudh", "Mannuru" ,"Network",
18000,"Male", 20),
                                new Employee(213, "Uma", "Sirisella", "User", 50000, "Male", 31),
                                new Employee(213, "Devi", "Rani", "User", 50000, "Female", 26),
                                new Employee(213, "Divya", "Utasad", "User", 50000, "Female", 27),
```

new Employee(213, "Ashu", "Kongala", "User", 50000, "Female", 28));

<u>**S No**</u>	<u>**Method name**</u>	<u>**Method description/situation**</u>	<u>**Output**</u>	<u>**Example Program**</u>
1	public static Collector<CharSequence, ?, String> joining()			String result1 = list.stream().map(Employee::getFirstName).collect(Collectors.joining());//No delimiter
2	public static Collector<CharSequence, ?, String> joining(CharSequence delimiter)			String result2 = list.stream().map(Employee::getFirstName).collect(Collectors.joining(","));//',' is demimiter
3	public static Collector<CharSequence, ?, String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)			String result3 = list.stream().map(Employee::getFirstName).collect(Collectors.joining("deleimter", "prefix", "suffix"));

4	public static <T, K> Collector<T, ?, Map<K, List<T>>> groupingBy(Function<? super T, ? extends K> classifier)			Map<String, List<Employee>> groupByDept = list.stream().collect(Collectors.groupingBy(Employee::getDeptName));
				System.out.println(groupByDept);

5	public static <T, K, A, D> Collector<T, ?,			Map<String, Long> groupByDept1 = list.stream().collect(Collectors.groupingBy(Employee::getDeptName, Collectors.counting()));

	Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier, Collector<? super T, A, D> downstream)			System.out.println(groupByDept1);
6	public static <T, K, D, A, M extends Map<K, D>> Collector<T, ?, M> groupingBy(Function<? super T, ? extends K> classifier, Supplier<M> mapFactory, Collector<? super T, A, D> downstream)			Map<String, Long> groupByDept2 = list.stream().collect(Collectors.groupingBy(Employee::getDeptName, LinkedHashMap::new, Collectors.counting()));
7	partitiningBy(p2)			
8	partitionBy(p1,p2)			

9	public static <T, U, A, R> Collector<T, ?, R>
	
	mapping(Function<? super T,? extends U> mapper,Collector<? super U,A,R> downstream)
		We can use these as downstream for grouping by methods also.		List<String> employeeNames = employeeList
				.stream()
				.collect(Collectors.mapping(Employee::getName, Collectors.toList()));

10	filtering()	To		
11	toMap()			
12	toList()			

13	toSet()			
14				
15				
16				
17				

Stream of Primitives:

- Java 8 offers the possibility to create streams out of three primitive types: **int, long and double.**
- As **Stream<T> is a generic interface**, and there is no way to use primitives as a type parameter with generics, three new special interfaces were created: **IntStream, LongStream, DoubleStream**.

```
IntStream intStream = IntStream.range(1, 3);
LongStream longStream = LongStream.rangeClosed(1, 3);
```

- ✓ The **range(int startInclusive, int endExclusive**) method creates an ordered stream from the first parameter to the second parameter. It increments the value of subsequent elements with the step equal to 1. The result doesn't include the last parameter, it is just an upper bound of the sequence.
- ✓ The rangeClosed(int startInclusive, int endInclusive) method does the same thing with only one difference, the second element is included. We can use these two methods to generate any of the three types of streams of primitives.
- ✓ Since Java 8, the Random(Random (Java SE 17 & JDK 17) (oracle.com))class provides a wide range of methods for generating streams of primitives.
- ✓ For example, the following code creates a DoubleStream, which has three elements:

```
Random random = new Random();
DoubleStream doubleStream = random.doubles(3);
```

Using the new interfaces alleviates unnecessary auto-boxing, which allows for increased productivity:

Primitive Streams:

1. String.chars() --- return IntStream
2. boxed() --- we can apply to any primitive streams to convert wrapper streams.
3. codePoints – it is similar like chars() but it give more feature like java.lang.Character
4. mapToObj
5. Stream.of
6. mapToInt

7. Stream.of()
8. Diff betweem map and flatmap

Lombok :

- @Builder
- @SuperBuilder – if we have IS-A relationship and if you want to get all the properties in the builder design pattern then you have ti use @SuperBuilde rin both child and parent class from v 1.87.2 version onword

Wrapper Stream:

Collectors Utility Classes:

Programs:

1. Convert character array to string in java

PermGem vs MetaSpace:

Reference link : Java Memory: PermGen vs MetaSpace | LinkedIn

Generic in java:

Before we have Arrays and it accepts homo generous type sand fixed.

- Let's say we have a student class and we need three different type of student id acceptable students (one could be int, one could be long, and another could string) how we can achieve this?
- Generally, we need to write three different Student class structures to achieve this. But using generics we can achieve this functionality with one class structure.

```
Class Student<T > {
private T studentId;
}
```

In above example we can pass T could be int or long or string or user defined data types as well.

Example:

Student<Integer> s1 = new Student();

Student<Long> s2 = new Student();

Student<String> s3 = new Student();

We have one class but we can able to create three different structure of student objects at run time with help of generics

- We can achieve type safety with help of generics.

<u>To make type safety in collections, they have introduced generic.</u>