**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?

* 1. 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 = () -> { };**
  2. 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>).

**Java 8 — Local Variables and Lambdas:**

Let’s break down how to consume local variable with lambda functions in java and the following scenarios will be covered:

1. Lambda consumes local variable initialized in a method
2. Local variable initialized in the Lambda function
3. Manipulating local variable inside the lambda function
4. Manipulating local variable outside the lambda function
5. Lambda functions consuming local reference variable
6. Declaring variables inside the lambda function
7. Using This in Lambda
8. Naming Convention in Lambda

**Lambda and Local Variables**

* Let’s take the case where a variable is being consumed by Lambda
* Let’s take a class (**Doge**) that consumes a Functional Interface (**PrintCoin**) which prints the **number of coins.**
* The Functional Interface look like this, which takes a single abstract method **print**()

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  } |

The Class Doge looks like this, we define the function as

|  |
| --- |
| () -> System.out.println("Hey Gurl! I see dem coins. I got "+coin+" coins"); |

which gets mapped to print()

|  |
| --- |
| class Doge{  public void getDoge(){  int coin = 100;  PrintCoin printCoin **= () -> System.out.println("Hey Gurl! I see dem coins. I got "+coin+" coins");**  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  The output will look like this  Hey Gurl! I see dem coins. I got 100 coins |

**Local variable initialized in a method**

Just like in the above example, **the local variable coin is consumed by the lambda,**

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  } |

|  |
| --- |
| class Doge{  public void getDoge(){  **int coin = 100; //local variable**  PrintCoin printCoin = () -> System.out.println("Hey Gurl! I see dem coins. I got "+coin+" **coins"); //lambda consumes it**  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  **Output**  Hey Gurl! I see dem coins. I got 100 coins |

**Local variable initialized in the Lambda function**

* Now let’s try and change the variable inside the lambda function.

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  public void getDoge(){  int coin; ---------------------🡪 This is local variable for method getDoge and it stored in it’s own thread stack  PrintCoin printCoin = () -> {  coin = 100; // Compile Time Error  System.out.println("Hey Gurl! I see dem coins. I got "+coin+" coins");  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  Output  java: local variables referenced from a lambda expression must be final or effectively final |

Whoa! What just happened?

Let’s break down all the questions scramming in our heads!

What is effectively final?

**Without having to explicitly say that a variable is final, i.e unlike**

final int number = 30;

where we have explicitly declared final. Incase of effectively final we won’t have to declare final keyword.

* Final variable insinuates that it’s a constant and will never change.
* **The same applies for effectively final**.
* Except the compiler now can understand that it’s final without us having to explicitly write the final keyword it.
* Which means after initialization the value will never change.

**Why do local variables need to be final or effectively final?**

* To ensure that no concurrency issues occur.

How can concurrency issues occur because of local variables?

* Let’s say multiple threads are working on the method.
* Maintaining a consistent value for the variable gets difficult.
* The value of the variable is **unpredicatble**.
* Depending on the thread executing the value changes.
* So to deal with concurrent operations the local variables **are final or effectively final.**

Does this occur for static and instance variables too (Instance Variable, Static Variable)?

NOPE!

Why does it not occur for Static or Instance Variables?

* Let’s talk about the memory storage of the variables,
* Instance variables are stored in the heap area.
* Static variables previously were stored in PermGen (method area) and now in Metaspace.
* Local variables are stored on the **stack**.

What does this have got to do?

* The stack reference is unique to each thread so the local variables are not **common to each thread**, whereas instance and static variables **are commonly available** and at all times their values will be **consistent for each thread.**
* In short, the location of where the variables are being placed is the **reason why static and instance variables don’t face the issue.**

**Manipulating local variable outside the lambda function**

Let’s have the lambda consume coin, but let’s manipulate the value outside the lambda as shown,

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  } |

**Example:**

|  |
| --- |
| class Doge{  public void getDoge(){  int coin = 100;  PrintCoin printCoin = () -> {  System.out.println("Hey Gurl! I see dem coins. I got coins" + coin); //lambda consumes coin  };  coin ++; //manipulating outside lambda function  System.out.println(coin);  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  **Output?**  java: local variables referenced from a lambda expression must be final or effectively final  Yuck! Again? |

**Why?**

* As stated above the local variables in lambdas are final or effectively final.
* Which ensures that the values consumed will always be **consistent** in a **multithreaded environment.**
* Okay!

What if the lambda does not consume the variable?

* Then it works like a normal local variable. As shown below.

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  public void getDoge(){  int coin = 100;  PrintCoin printCoin = () -> {  System.out.println("Hey Gurl! I dont want no coins");  };  coin ++; // not consumed by lambda  System.out.println("Printing coins that are not consumed by lambda = "+ coin);  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  **The output looks like this,**  Printing coins that are not consumed by lambda = 101  Hey Gurl! I dont want no coins |

**Manipulating Local Variables inside Lambda Functions**

Let’s manipulate the value of coin inside the lambda function now,

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  public void getDoge(){  int coin = 100;  PrintCoin printCoin = () -> {  **coin ++;** //manipulating inside lambda function  System.out.println("Hey Gurl! I dont want no coins");  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  Output?  java: local variables referenced from a lambda expression must be final or effectively final  Why?  You know it! It’s the same reason all over! EFFECTIVELY FINAL/FINAL, CONCURRENCY Blah Blah! |

**Lambda functions consuming local reference variable**

* Final refers that once initialized we can’t reinitialize.
* The same works for reference we can initialize it once **but no reinitialization is allowed**.

|  |
| --- |
| import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  public void getDoge(){  int coin = 100;  List<Integer> coinsBought = new ArrayList();  PrintCoin printCoin = () -> {  coinsBought = Arrays.asList(80,20); //compile time error  System.out.println("Hey Gurl! I dont want no coins");  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  **Output is the same ole,**  **java: local variables referenced from a lambda expression must be final or effectively final** |

* No reinitialization allowed for finals/effectively final!

How to manipulate the local variables of a method, that get’s consumed by Lambda function?

* Can use Atomic References eg, AtomicIntegers which ensures concurrency. Although we shouldn’t be manipulating local variables inside lambdas.

Can we add values to references?

Don’t get confused!

Final ensures that the reference never changes, the value can change! As shown below.

|  |
| --- |
| import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  public void getDoge(){  int coin = 100;  List<Integer> coinsBought = new ArrayList();  PrintCoin printCoin = () -> {  coinsBought.add(80); //allowed  coinsBought.add(20);  int value = coin + coinsBought.stream().reduce(0 , (coin1,coin2) -> coin1 + coin2);  System.out.println(value +" Doge coins in my bag!");  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  **The output looks like this,**  200 Doge coins in my bag!  Yay!! |

Declaring Variables Inside Lambda Function

Can we declare variables inside Lambda Function? HELL YEAH!

In the above example, we had done it too! Let’s dig down

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  public void getDoge(){  int coin = 100;  List<Integer> coinsBought = new ArrayList();  PrintCoin printCoin = () -> {  coinsBought.add(80);  coinsBought.add(20);  int value = coin + coinsBought.stream().reduce(0 , (coin1,coin2) -> coin1 + coin2); //declared a variable inside lambda  System.out.println(value +" Doge coins in my bag!");  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  **Output,**  200 Doge coins in my bag!  Can we manipulate the local variable declared inside the lambda function? YES YES AND YES! It’s not final! |

**Example:**

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  public void getDoge(){  int coin = 100;  List<Integer> coinsBought = new ArrayList();  PrintCoin printCoin = () -> {  coinsBought.add(80);  coinsBought.add(20);  int value = coin + coinsBought.stream().reduce(0 , (coin1,coin2) -> coin1 + coin2);  value ++;  System.out.println(value +" Doge coins in my bag!");  value ++; //can manipulate as it's not final!  System.out.println(value +" Doge coins ++");  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  **Output,**  201 Doge coins in my bag!  202 Doge coins ++  Similarly, let’s see with references, |

**Example:**

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  public void getDoge(){  PrintCoin printCoin = () -> {  List<Integer> coinsBought = new ArrayList();  coinsBought.add(80);  coinsBought.add(20);  System.out.println(" Doge coins bought = "+coinsBought);  coinsBought = Arrays.asList(100,200); //no error  System.out.println(" Doge coins bought = "+coinsBought);  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  Output,  Doge coins bought = [80, 20]  Doge coins bought = [100, 200]  Worked! |

**Example:**

|  |
| --- |
| This in Lambda  This refers to the current outer reference. Unlike Anonymous classes  @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  int coin = 30; //this referes to current outer value  public void getDoge(){  int coin = 20;  PrintCoin printCoin = () -> {  System.out.println(" Doge coins bought = "+ this.coin);  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  Output  Doge coins bought = 30  How to access the local variable? Remove this! Easy! |

**Example:**

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  int coin = 30;  public void getDoge(){  int coin = 20;  PrintCoin printCoin = () -> {  System.out.println(" coin = "+ coin);  System.out.println(" this.coin = "+ this.coin);  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  Output,  coin = 20  this.coin = 30  Naming Convention in Lambda  The local variable name defined inside the method and the local variable name defined inside the lambda function cannot be the same! |

**Example:**

|  |
| --- |
| @FunctionalInterface  interface PrintCoin{  void print();  }  class Doge{  int coin = 30;  public void getDoge(){  int coin = 20;  PrintCoin printCoin = () -> {  int coin = 40; //cannot be the same  };  printCoin.print();  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  Output  java: variable coin is already defined in method getDoge()  The local variable name defined inside the method and the parameter name of the lambda function cannot be the same! |

**Example:**

|  |
| --- |
| package com.company;  import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  @FunctionalInterface  interface PrintCoin{  void print(int value);  }  class Doge{  int coin = 30;  public void getDoge(){  int coin = 20;  PrintCoin printCoin = (coin) -> { //cannot define same name  };  printCoin.print(200);  }  public static void main(String []args){  Doge doge = new Doge();  doge.getDoge();  }  }  Output  java: variable coin is already defined in method getDoge()  You’ve aced Lambdas yay! |

**Important:**

|  |
| --- |
| The answer is simply that your variable goes out of **scope at** the end of the method. **This is easily solved with effectively final variables as the compiler just copies the value into the lambda**. Since the code in the lambda expression can also be run outside of the method (where the modifiable variable is garbage collected already) this won't work. You also can't expect the compiler to somehow make a copy of your variable and then dynamically change it when it's modified outside of your lambda expression. I hope that clears it up. |

**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();  } |

1. **Comparable**<T> :

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

1. **Comparator<T>:**

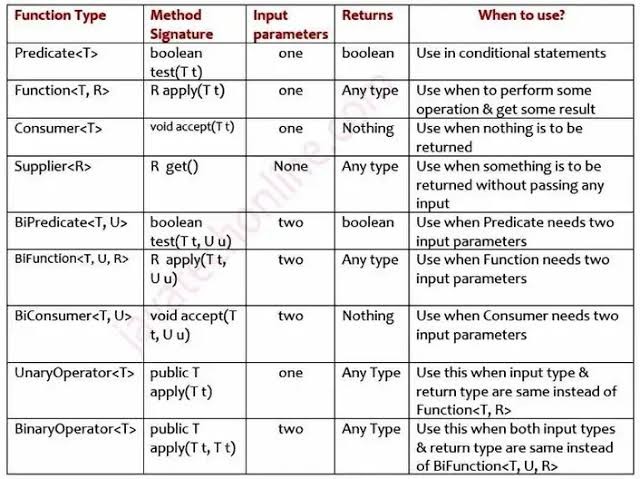
|  |
| --- |
| @FunctionalInterface  public interface Comparator<T> {  int compare(T o1, T o2);  } |

1. **Callable<V>:**

|  |
| --- |
| @FunctionalInterface  public interface Callable<V> {  V call() throws Exception;  } |

1. **ActionListener**

**Java 8 predefined functional interfaces:**



**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 l2 {  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  }  } |

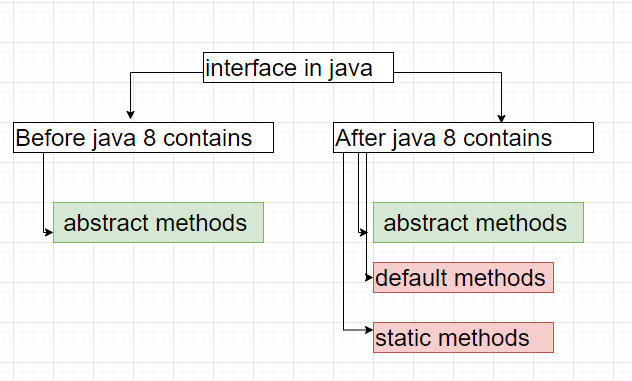
**Difference between static and default methods in java8 interface:**

**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’.

**There are two main difference between Abstract Class and Interface in java 8.**

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:

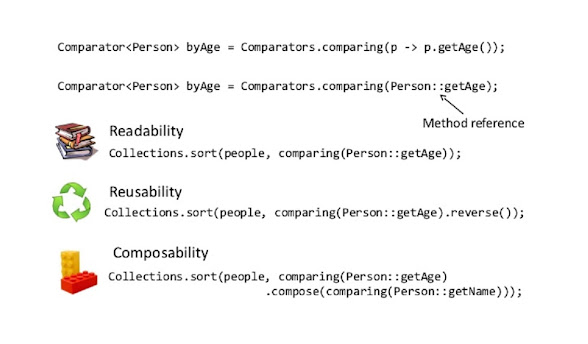
* A static method (ClassName::methodName) like Person::getAge
* An instance method of a particular object (instanceRef::methodName) like System.out::println
* A super method of a particular object (super::methodName)
* An instance method of an arbitrary object of a particular type (ClassName::methodName)—existing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

**Reference to an Instance Method of an Arbitrary Object of a Particular Type**

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).



**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.

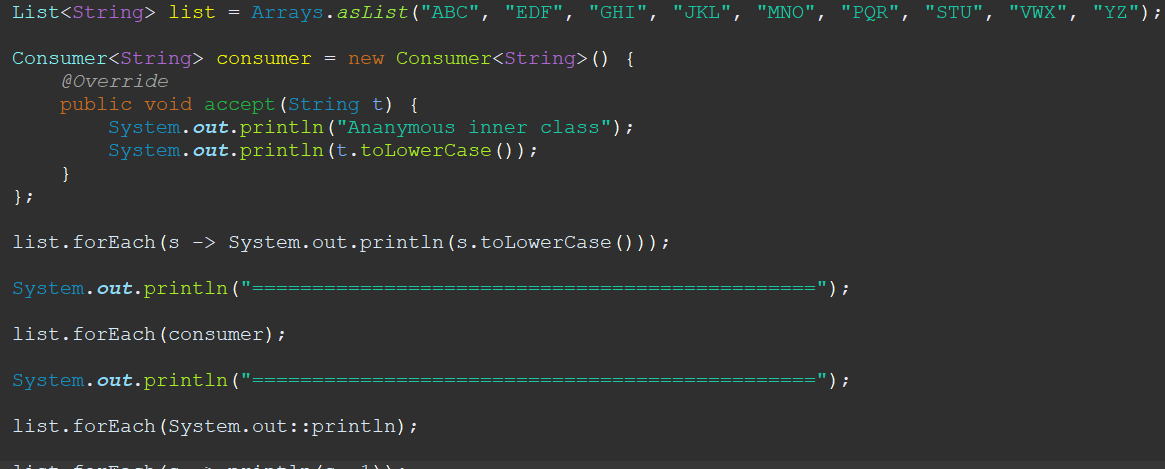
|  |
| --- |
| List<A> a = Arrays.asList(new A(2), new A(3));  a.stream().filter(b -> b.is()).forEach(System.out::println);;  a.stream().filter(A::is); |

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

**Is Method References being replacements of LamdaExpresions:**

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

But we will use Method References, we can use

****

**Introduction to Method Reference**

* **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);  }  }; |

1. **Using lambda expression**

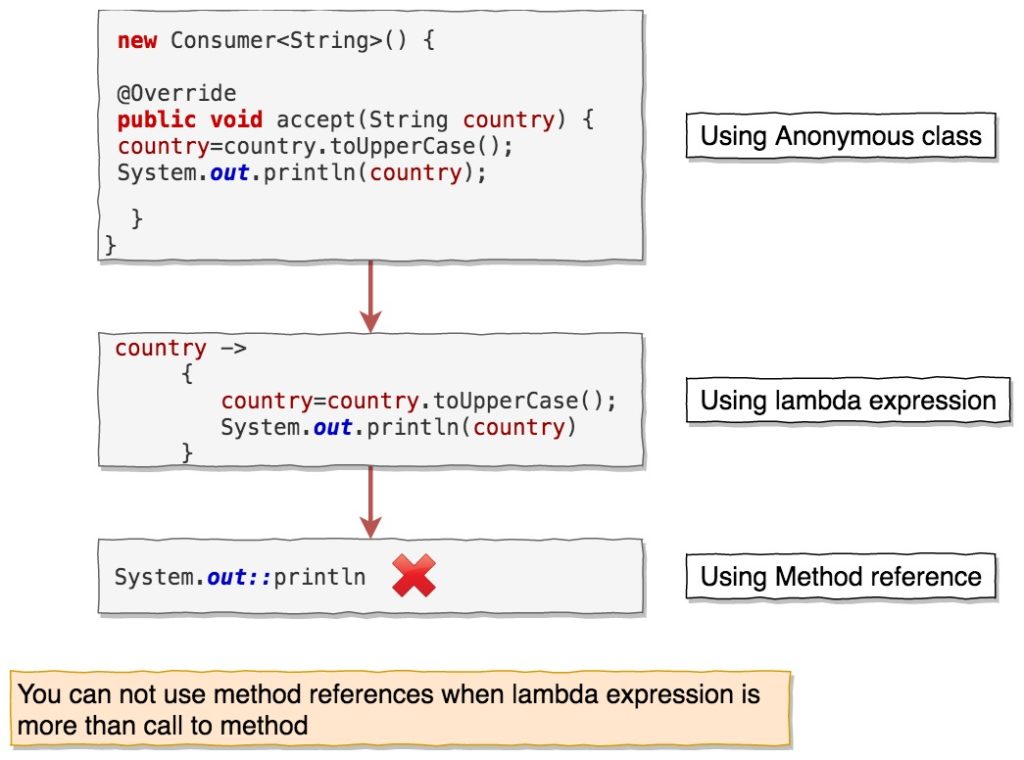
|  |
| --- |
| Consumer<string> consumer2 = country -> System.out.println(country); |

1. **Using method reference**

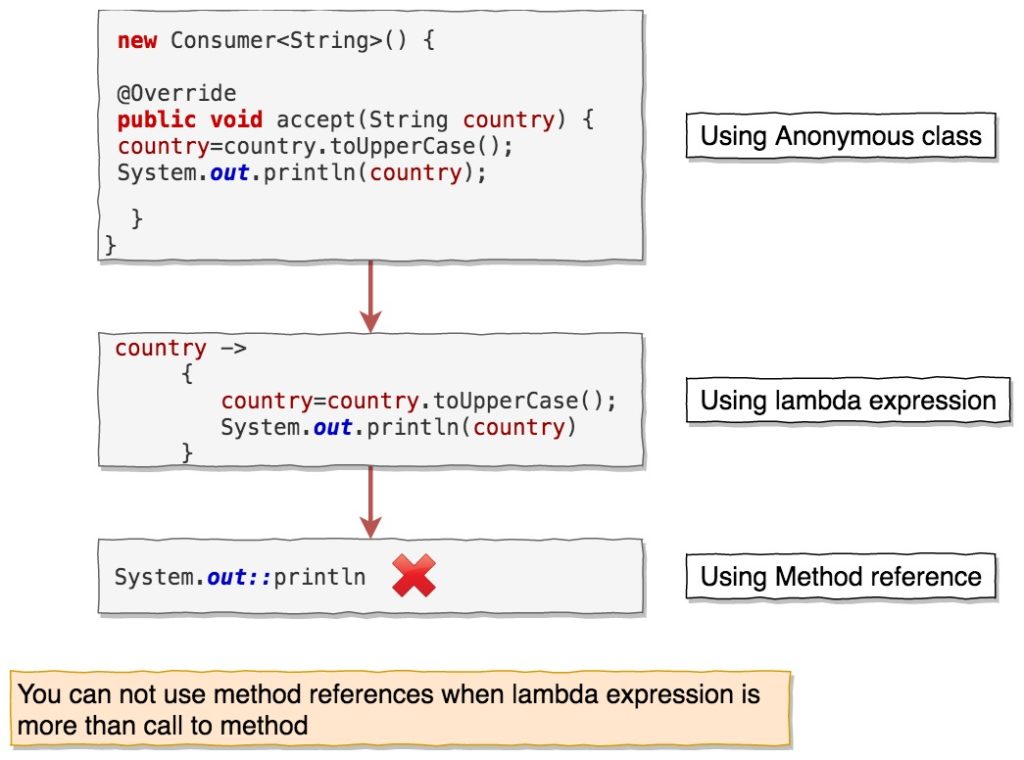
|  |
| --- |
| Consumer<string> consumer3 = System.out::println; |

* 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.



You cannot use method reference as below:



**Types of method references:**

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().

1. **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

|  |
| --- |
| **Example :** List<String> list = Ararys.asList(“ABC”, “DEF”);  list.stream().map(String::toLowerCase).collect(Collectors.toList());  **Another Example:**  List<Person> persons = Ararys.asList(new Person(), new Person());  persons.stream().map(Person::isBatcher).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

obje ctType::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/**](https://java2blog.com/java-8-method-reference/)

**Streams:**

**Streams Introduction:**

**Stream is collecting the complete data part by part**

* At the granular level, the difference between a Collection and a Stream is when the things are computed.
* A Collection is an in-memory data structure that holds all the data structure’s values. Every element in the Collection has to be computed before it can be added to the Collection.
* While a Stream is conceptually a pipeline in which elements are computed on demand.

|  |
| --- |
| List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  // Using a Stream to filter even numbers and then double them  List<Integer> evenNumber = numbers.stream()  **.filter(**n -> n % 2 == 0) // Filter even numbers  .toList(); // Collect the results into a new list  System.out.println("Even Numbers List: " + evenNumber); // [2, 4, 6, 8, 10] |

* Stream operations are either **intermediate or terminal**. The terminal operations return a result of a certain type, and intermediate operations return the stream itself so we can chain multiple methods in a row to perform the operation in multiple steps.
* Streams are created on a source, e.g. a java.util.Collection like List or Set. The Map is not supported directly, we can create a stream of map keys, values or entries.
* Stream operations can either be executed sequentially or in parallel. when performed parallelly, it is called a parallel stream.

**Based on the above points, we can say that a Stream is:**

* Designed for lambdas or functional programming
* Not a data structure to store objects
* Do not support indexed access
* Can easily be aggregated as arrays or lists
* Lazy access supported
* Parallelizable

**Creating Streams:**

|  |  |
| --- | --- |
| Method name | Example |
| Stream.of() | Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);  stream.forEach(p -> System.out.println(p)); |
| Stream.of(array) | Stream<Integer> stream = Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9} );  stream.forEach(p -> System.out.println(p)); |
| List.stream() | List<Integer> list = new ArrayList<Integer>();  for(int i = 1; i< 10; i++){  list.add(i);  }  Stream<Integer> stream = list.stream();  stream.forEach(p -> System.out.println(p)); |
| Stream.generate() or Stream.iterate() | Stream<Integer> randomNumbers = Stream  .generate(() -> (new Random()).nextInt(100));  randomNumbers.limit(20).forEach(System.out::println); |
| Stream of String chars or tokens | IntStream stream = "12345\_abcdefg".chars();  stream.forEach(p -> System.out.println(p));  //OR  Stream<String> stream = Stream.of("A$B$C".split("\\$"));  stream.forEach(p -> System.out.println(p)); |
| concat() | // Creating two Streams  Stream<String> stream1 = Stream.of("Geeks", "for");  Stream<String> stream2 = Stream.of("GeeksQuiz", "GeeksforGeeks");    // concatenating both the Streams  // with Stream.concat() function  // and displaying the result  Stream.concat(stream1, stream2)  .forEach(element -> System.out.println(element)); |
| empty() | // Creating an empty Stream  Stream<String> stream = Stream.empty();    // Displaying elements in Stream  stream.forEach(System.out::println); |
| Files.lines | Path path = Paths.get("C:\\file.txt");  Stream<String> streamOfStrings = **Files**.**lines**(path);  Stream<String> streamWithCharset =  **Files.lines(path, Charset.forName("UTF-8"));** |

**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.

**How many ways we can create Stream object?**

**1. Stream of Array:**

* 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(); |

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

**2.Stream.iterate():**

* 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.

**3.Stream of String:**

* 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"); |

**4. Stream of File:**

* 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:

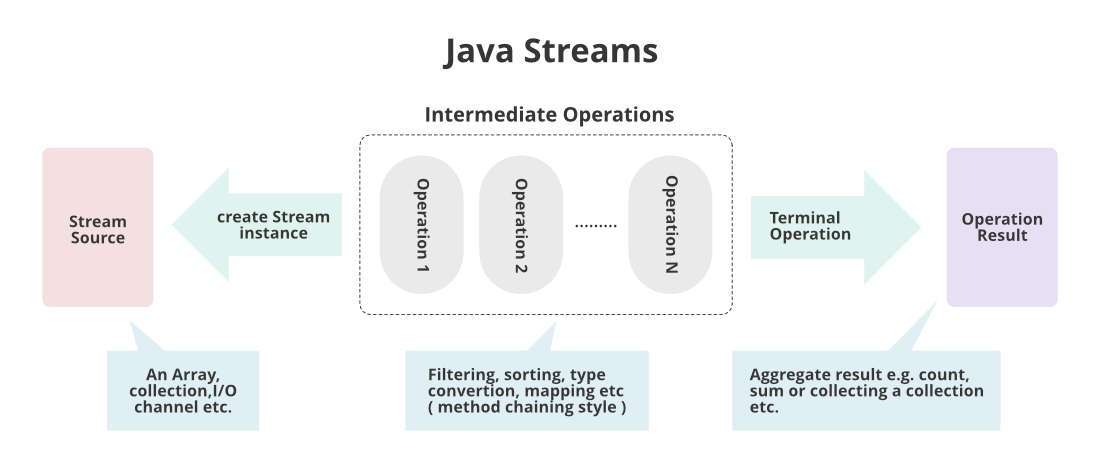
|  |
| --- |
| long size = list.stream().skip(2).map(element -> {  wasCalled();  return element.substring(0, 3);  }).count(); |

* 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.



**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:

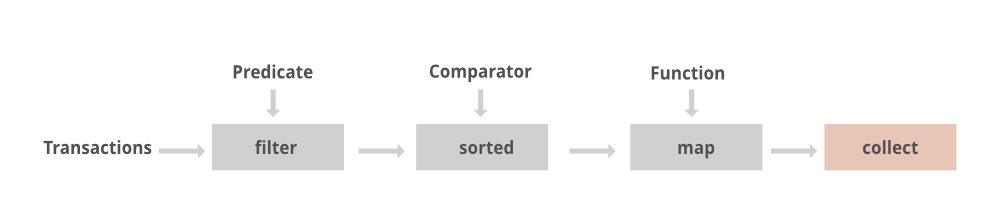
|  |
| --- |
| Stream<Product> streamOfCollection = productList.parallelStream();  boolean isParallel = streamOfCollection.isParallel();  boolean bigPrice = streamOfCollection  .map(product -> product.getPrice() \* 12)  .anyMatch(price -> price > 200); |

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:**

**Intermediate Methods:** Java 8 Stream intermediate operations return another Stream which allows you to call multiple operations in the form of a **query**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S No | Method name | Method description/situation | Method Input/Functional Interface Type | Example Program |
| 1 | stream() | Returns a sequential stream considering collection as its source. |  | List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");  List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList()); |
| 2 | parallelStream() | Returns a parallel Stream considering collection as its source. |  |  |
| 3 | Stream.filter() | The filter() method accepts a Predicate to filter all elements of the stream. This operation is intermediate, enabling us to call another stream operation (e.g. forEach()) on the result. |  | memberNames.stream().filter((s) -> s.startsWith("A"))  .forEach(System.out::println); |
| 4 | Stream.map() | The map() intermediate operation converts each element in the stream into another object via the given function.  The following example converts each string into an UPPERCASE string. But we can use map() to transform an object into another type as well. |  | memberNames.stream().filter((s) -> s.startsWith("A"))  .map(String::toUpperCase)  .forEach(System.out::println); |
| 5 | Stream.flatmap() | Reference Link: <https://howtodoinjava.com/java8/stream-flatmap-example/> |  |  |
| 6 | Stream.sorted() | The sorted() method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom Comparator.  Please note that the sorted() method only creates a sorted view of the stream without manipulating the ordering of the source Collection. In this example, the ordering of string in the memberNames is untouched. |  | memberNames.stream().sorted()  .map(String::toUpperCase)  .forEach(System.out::println); |
| 7 | peek() | **Note:** From java9 onwards, for some of terminal operations like count(), intermediate operations are not getting triggered until it demands to do. So be case full when you use peek() method. |  | List<String> genre = new ArrayList<String>(Arrays.asList("rock", "pop", "jazz", "reggae"));  System.out.println(genre.stream().map(s -> s.replace("r", "x"))  .peek(s -> System.out.println(s))  .filter(s -> s.indexOf("x") == 0).count()); |
| 8 | limit() | The limit(n) method is another intermediate operation that returns a stream not longer than the requested size. As before, the n parameter can’t be negative. |  | Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  .filter(i -> i % 2 == 0)  .limit(2)  .forEach(i -> System.out.print(i + " "))  As is the case with the skip() operation, limit() is also a stateful operation since it has to keep the state of the items that are being picked up.  But unlike skip(), which consumes the entire stream, as soon as limit() reaches the maximum number of items, it doesn’t consume any more items and simply returns the resulting stream. Hence, we say that limit() is a short-circuiting operation.  When working with infinite streams, limit() can be very useful for truncating a stream into a finite one:  Stream.iterate(0, i -> i + 1)  .filter(i -> i % 2 == 0)  .limit(10)  .forEach(System.out::println); |
| 9 | skip() | The skip(n) method is an intermediate operation that discards the first n elements of a stream. The n parameter can’t be negative, and if it’s higher than the size of the stream, skip() returns an empty stream. |  | Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  .filter(i -> i % 2 == 0)  .skip(2)  .forEach(i -> System.out.print(i + " "));  private static List<Integer> getEvenNumbers(int offset, int limit) {  return Stream.iterate(0, i -> i + 1)  .filter(i -> i % 2 == 0)  .skip(offset)  .limit(limit)  .collect(Collectors.toList());  } |
| 10 | distinct() | To remove the duplicacy from the elements of a stream, we can use the distinct() method.  Based the object equals and hascode method.  Note: If we want to apply distinct() method to remove list of custom objects then we must override equals and hascode methods , based on that comparision only distinct() method weill decide which is duplicate. |  | List<Integer> numbers = Arrays.asList(21, 21, 32, 14, 19, 19, 10, 10);  System.out.println("Distinct list of numbers: ");  // removing duplicate numbers from the list  numbers.stream()  .distinct()  .forEach(System.out::println);1 |
| 11 |  |  |  |  |

**What is thedifference between map() and flatmap() stream method:**

map() produces a Stream consisting of the results of applying the Functional Function interface refrence/lamda expresion

Example : List<String> myList = Stream.of("a", "b") .map(String::toUpperCase) .collect(Collectors.toList());

By applying the toUpperCase() method to the elements of the input Stream

map() works pretty well in such a simple case. But what if we have something more complex, such as a list of lists as an input?

List<List<String>> list = Arrays.asList(Arrays.asList("a"), Arrays.asList("b"));

**output**: [[a,b]]

System.out.println(list.stream() .flatMap(Collection::stream).collect(Collectors.toList()));

**output**: The result of such a snippet will be flattened to [a, b].

The flatMap() method first flattens the input Stream of Streams to a Stream of Strings (for more about flattening, see this article). Thereafter, it works similarly to the map() method.

**What will heppend when we give different type of streams for flatmap():**

|  |
| --- |
| List<String> stringList1 = Arrays.asList("a");  List<Integer> intList1 = Arrays.asList(1, 2, 3, 4, 5, 6);  List<Integer> intListt2 = Arrays.asList(1, 2, 3, 4, 5, 6);  List<List<? extends Object>> differentTyPesOfList = Arrays.asList(intList1, intListt2, stringList1);  **List<? extends Object>** result1 = differentTyPesOfList.stream().flatMap(f -> f.stream()).collect(Collectors.toList()); |

Java 8 Stream.flatMap() method is used to flatten a Stream of collections to a Stream of objects. During the flattening operation, the objects from all the collections in the original Stream are combined into a single collection

**Stream<Collection<Item>> —-> flatMap() —-> Stream<Item>**

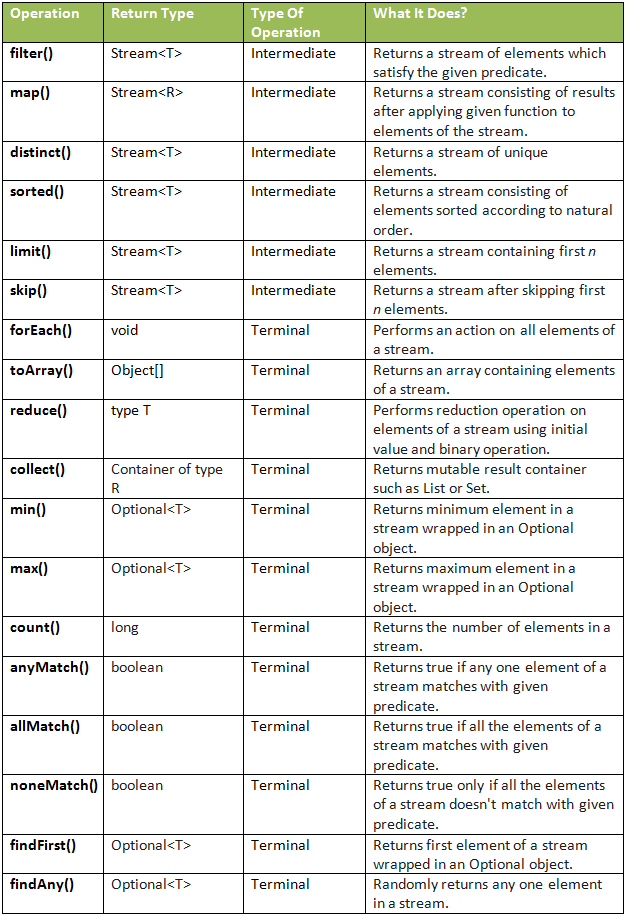
|  |
| --- |
| List<List<Integer>> listOfLists = Arrays.asList(  Arrays.asList(1, 2, 3),  Arrays.asList(4, 5),  Arrays.asList(6, 7, 8)  );  List<Integer> flattenedList = listOfLists.stream()  .flatMap(list -> list.stream()) // Flattening step  .toList();  //Prints [1, 2, 3, 4, 5, 6, 7, 8]  System.out.println("Flattened list: " + flattenedList); |

**Interview Questions example for flatmap:**

|  |
| --- |
| List<List<String>> myList = Arrays.asList(  Arrays.asList("1", "3", "99", "2", "14", "2"),  Arrays.asList("111", "7", "199", "92", "14", "20"),  Arrays.asList("10", "31", "9", "21", "243", "432"),  Arrays.asList("31", "13", "999", "5", "41", "222"),  Arrays.asList("10", "31", "401", "42", "11", "77"));    List<String> asList = Arrays.asList("S");    List<List<List<String>>> myList1 = Arrays.asList(Arrays.asList(  Arrays.asList("1", "3", "99", "2", "14", "2"),  Arrays.asList("111", "7", "199", "92", "14", "20"),  Arrays.asList("10", "31", "9", "21", "243", "432"),  Arrays.asList("31", "13", "999", "5", "41", "222"),  Arrays.asList("10", "31", "401", "42", "11", "77")));  List<List<List<String>>> asList2 = Arrays.asList(Arrays.asList(  Arrays.asList("a","b"), Arrays.asList("c","d"), Arrays.asList("e","f")));    //  // List<String> onlyab = asList2.stream()  // .flatMap(Collection::stream)  // .findFirst().get();    // onlyab.forEach(System.out::println);    String onlyaValue = asList2.stream()  .flatMap(c -> c.stream())  .flatMap(Collection::stream)  .findFirst().get();    System.out.println(onlyaValue);      Predicate<Integer> oddnumber = i -> i % 2 != 0;    //arr = [[],[],[]]; --- > [1,];  //arr.forEach(list ->  List<Integer> collect = myList.stream()  //.flatMap(Collection::stream)  .flatMap(list -> list.stream().map(Integer::valueOf).filter(oddnumber))  .collect(Collectors.toList());    System.out.println(collect); |

**Terminal Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S No | Method name | Method description/situation | Method Input/Functional Interface Type | Example Program |
| 1 | Stream.forEach() | The forEach() method helps iterate over all stream elements and perform some operation on each of them. The operation to be performed is passed as the lambda expression. |  | memberNames.forEach(System.out::println); |
| 2 | Stream.collect() | The collect() method is used to receive elements from steam and store them in a collection. |  | List<String> memNamesInUppercase = memberNames.stream().sorted()  .map(String::toUpperCase)  .collect(Collectors.toList());  System.out.print(memNamesInUppercase);  Program Output:  [AMAN, AMITABH, LOKESH, RAHUL, SALMAN, SHAHRUKH, SHEKHAR, YANA] |
| 3 | Stream.match() | Various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a boolean result. |  | boolean matchedResult = memberNames.stream()  .anyMatch((s) -> s.startsWith("A"));    System.out.println(matchedResult); //true    matchedResult = memberNames.stream()  .allMatch((s) -> s.startsWith("A"));    System.out.println(matchedResult); //false    matchedResult = memberNames.stream()  .noneMatch((s) -> s.startsWith("A"));    System.out.println(matchedResult); //false |
| 4 | Stream.count() | The count() is a terminal operation returning the number of elements in the stream as a long value. |  | long totalMatched = memberNames.stream()  .filter((s) -> s.startsWith("A"))  .count();  System.out.println(totalMatched); //2 |
| 5 | Stream.reduce() | The reduce() method performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value. | T reduce(T identity, **BinaryOperator<T> accumulator**); | Optional<String> reduced = memberNames.stream()  .reduce((s1,s2) -> s1 + "#" + s2);    reduced.ifPresent(System.out::println);  Program Output:  Amitabh#Shekhar#Aman#Rahul#Shahrukh#Salman#Yana#Lokesh |
| 6 | forEachOrdered() | forEachOrdered will processes the elements of the stream in the order specified by its source, regardless of whether the stream is sequential or parallel. |  | Stream.of("AAA","BBB","CCC").parallel().forEach(s->System.out.println("Output:"+s));// not guarantee the order  Stream.of("AAA","BBB","CCC").parallel().forEachOrdered(s->System.out.println("Output:"+s));//  The second line will always output  Output:AAA  Output:BBB  Output:CCC |
| 7 | toArray() |  |  | 1. String lines = "I Love Java 8 Stream!";   // split by space, uppercase, and convert to Array  String[] result = Arrays.stream(lines.split("\\s+"))  .map(String::toUpperCase)  .toArray(String[]::new);   1. int[] num = {1, 2, 3, 4, 5};   Integer[] result = Arrays.stream(num)  .map(x -> x \* 2)  .boxed()  .toArray(Integer[]::new);   1. int[] stream1 = IntStream.rangeClosed(1, 5).toArray(); 2. Stream<Integer> stream2 = Stream.of(1, 2, 3, 4, 5);   int[] result = stream2.mapToInt(x -> x).toArray(); |
| 8 | min() |  |  | // Finding Smallest Number  Optional<Integer> minNumber = numbers.stream()  .min((i, j) -> i.compareTo(j));  System.out.println("Smallest number: " |
| 9 | max() |  |  | // Finding Largest Number  Optional<Integer> largestNumber = numbers.stream()  .max((i, j) -> i.compareTo(j));  System.out.println("Largest number: "  + largestNumber.get()); |
| 10 | findAny() | As the name suggests, the findAny() method allows us to find any element from a Stream. We use it when we’re looking for an element without paying an attention to the encounter order |  | List<String> list = Arrays.asList("A","B","C","D");  Optional<String> result = list.stream().findAny(); |
| 11 |  |  |  |  |



**Short-circuit Operations:**

* Though stream operations are performed on all elements inside a collection satisfying a Predicate, it is often desired to break the operation whenever a matching element is encountered during iteration. In external iteration, we will do with the if-else block. In the internal iterations such as in streams, there are certain methods we can use for this purpose.
* For example anyMatch , allMatch , noneMatch , findFirst and findAny are short circuiting terminal operations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S No | Method name | Method description/situation | Method Syntax | Example Program |
| 1 | Stream.anyMatch() | The anyMatch() will return true once a condition passed as predicate satisfies. Once a matching value is found, no more elements will be processed in the stream.  In the given example, as soon as a String is found starting with the letter 'A', the stream will end and the result will be returned. |  | boolean matched = memberNames.stream()  .anyMatch((s) -> s.startsWith("A"));  System.out.println(matched); //true |
| 2 | Stream.findFirst() | The findFirst() method will return the first element from the stream and then it will not process any more elements. |  | String firstMatchedName = memberNames.stream()  .filter((s) -> s.startsWith("L"))  .findFirst()  .get();  System.out.println(firstMatchedName); //Lokesh |
| 3 | limlit() |  |  |  |

**Parallel Streams Methods:**

* With the Fork/Join framework added in Java SE 7, we have efficient machinery for implementing parallel operations in our applications.
* But implementing a fork/join framework is a complex task, and if not done right; it is a source of complex multi-threading bugs that have the potential to crash the application. With the introduction of internal iterations, we got the possibility of operations to be done in parallel more efficiently.
* To enable parallelism, all we have to do is to create a parallel stream, instead of a sequential stream. And to our surprise, this is really very easy.
* In any of the above-listed stream examples, anytime we want to do a particular job using multiple threads in parallel cores, all we have to call **parallelStream()** method instead of **stream()** method.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S No | Method name | Method description/situation | Method Syntax | Example Program |
| 1 | Stream. parallelStream() | In any of the above-listed stream examples, anytime we want to do a particular job using multiple threads in parallel cores, all we have to call parallelStream() method instead of stream() method. |  | List<Integer> list = new ArrayList<Integer>();  for(int i = 1; i< 10; i++){  list.add(i);  }  //Here creating a parallel stream  Stream<Integer> stream = list.parallelStream();  Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);  System.out.print(evenNumbersArr);); //true |

**Optional Class and it’s Methods:**

* The Optional class in Java 8 is a container object which is used to contain a value that might or might not be present.
* Optional object is used to represent null with absent value.
* This class has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values.
* It was introduced as a way to help reduce the number of NullPointerExceptions that occur in Java code. It is a part of the java.util package and was added to Java as part of Java 8.
* Optional is a container that either contains a non-null value or nothing(empty Optional ).
* The Optional class is an implementation of the Null Object pattern, which is a design pattern that was designed to reduce the number of null checks in code. With Optional, you can write cleaner and more concise code that is easier to maintain.
* One of the key benefits of using Optional is that it forces you to handle the case where the value is absent.
* This means that you are less likely to miss important checks in your code and reduces the risk of **NullPointerException**.
* If a value is not present, you can either provide a default value or throw an exception.

**Example (Problem with null):**

|  |
| --- |
| public class Main {  public static void main(String[] args) {  Student Student = getStudentWithName("hamza");  System.out.println(Student.getName());//NPE  }  public static Student getStudentWithName(String name ){  // lets suppose that our database contain only 2 students ahmed and hamza .  if (name.equals("hamza") || name.equals("ahmed")) {  return new Student(name , 22 , "Morocco");  } else {  **return null ;**  }  }  } |

**Explanation:**

* It’s obvious that **getStudentWithName**() can return a null if there is no student with the given name => NullPointerException will occur when we call the getName() method (student = null ).

To handle this case, we should implement some form of conditional logic based on the result returned by the method**.**

|  |
| --- |
| public class Main {  public static void main(String[] args) {  Student Student = getStudentWithName("hamza");  if(student != null){  System.out.println(Student.getName());  }else {  System.out.println("no Student with the given name ");  }  }  public static Student getStudentWithName(String name ){  // lets suppose that our database contain only 2 students ahmed and hamza .  if (name.equals("hamza") || name.equals("ahmed")) {  return new Student(name , 22 , "Morocco");  } else {  return null ;  }  }  } |

**To avoid NullPointerException problems, let’s examine what the Optional API offers.**

* The Optional class provides several methods that can be used to interact with the value that is stored inside the Optional. For example, the **isPresent()** method can be used to check if a value is present or not. If a value is present, you can use the **get()** method to retrieve it.

Now let’s try to refactor our exemple using Optional API .

|  |
| --- |
| public class Main {  public static void main(String[] args) {  Optional<Student> student = Optional.ofNullable(getStudentWithName("hamza"));  if (student.**isPresent**()) {  System.out.println(student.get().getName());  } else {  System.out.println("Student is not present");  }  }  public static Student getStudentWithName(String name ){  // lets suppose that our database contain only 2 students ahmed and hamza .  if (name.equals("hamza") || name.equals("ahmed")) {  return new Student(name , 22 , "Morocco");  } else {  return null ;  }  }  } |

**Make your code more concise using orElse() , orElseThrow() .**

1. orElse():

**orElse**()is a method in the Java Optional class that is used to return the value wrapped in the Optional, if it is present, or a default value if it is not. It takes a default value as an argument and returns it if the Optional is empty.

|  |
| --- |
| public class Main {  public static void main(String[] args) {  Student student = Optional.ofNullable(getStudentWithName("hamza")).**orElse**(new Student("no one", 0, "Unknown"));  System.out.println(student.getName());  }  public static Student getStudentWithName(String name ){  // lets suppose that our database contain only 2 students ahmed and hamza .  if (name.equals("hamza") || name.equals("ahmed")) {  return new Student(name , 22 , "Morocco");  } else {  return null ;  }  }  } |

1. orElseThrow():

**orElseThrow()** is a method in the Java Optional class that is used to throw an exception if the Optional is empty. Unlike orElse, which returns a default value, orElseThrow allows you to throw a custom exception.

|  |
| --- |
| public class Main {  public static void main(String[] args) throws StudentNotFoundException {  Student student = Optional.ofNullable(getStudentWithName("fs")).orElseThrow(()-> new StudentNotFoundException("the Student is not Present "));  System.out.println(student.getName());  }  public static Student getStudentWithName(String name ){  // lets suppose that our database contain only 2 students ahmed and hamza .  if (name.equals("hamza") || name.equals("ahmed")) {  return new Student(name , 22 , "Morocco");  } else {  return null ;  }  }  } |

Optional class in Java 8 is a useful tool for reducing the number of **NullPointerExceptions** in your code. It provides a clean and concise way to handle values that **might or might not be present** and **reduces** the risk of missing important checks in your code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S No | Method name | Method description/situation | Method Syntax | Example Program |
| Optional.ofNullable(T) |  | To use the Optional class, simply wrap your object in an Optional object like this: Optional<T> optional = Optional.ofNullable(T). You can then use the various methods provided by Optional to handle the cases where the value is present or not. For example, the isPresent() method returns true if the value is present, and false if it is not. |  |  |
| Optional.of(T) |  | This method behave like the ofNullable() function it will return an Optional for the given value the only difference is that the of() method does not allow to construct an Optional object around a null value . |  | Optional<Student> student = Optional.of(null);  // above line will throw a NullPointerException . |
| Optional.empty() |  | empty()  this method will simply generate an empty Optional (no value ). |  | Optional<T> empty = Optional.empty(); |
| isPresent() |  |  |  | Optional<String> opt = Optional.of("baeldung");  opt.ifPresent(name -> System.out.println(name.length())); |
| get() |  |  |  | Optional<String> opt = Optional.of("baeldung");  String name = opt.get();  Optional<String> opt = Optional.ofNullable(null);  String name = opt.get(); |
| orElse() |  |  |  | String name = Optional.ofNullable(nullName).orElse("john"); |
| orElseGet() |  |  |  | Optional.ofNullable(nullName).orElseGet(() -> "john"); |
| orElseThrow() |  |  |  | String nullName = null;  String name = Optional.ofNullable(nullName).orElseThrow(  IllegalArgumentException::new); |
| isEmpty() |  |  |  | Optional<String> opt = Optional.of("Baeldung");  assertFalse(opt.isEmpty());  opt = Optional.ofNullable(null);  assertTrue(opt.isEmpty( |
| filter() |  |  |  | Integer year = 2016;  Optional<Integer> yearOptional = Optional.of(year);  boolean is2016 = yearOptional.filter(y -> y == 2016).isPresent();  assertTrue(is2016);  boolean is2017 = yearOptional.filter(y -> y == 2017).isPresent();  assertFalse(is2017);  return Optional.ofNullable(modem2)  .map(Modem::getPrice)  .filter(p -> p >= 10)  .filter(p -> p <= 15)  .isPresent(); |
| Transforming Value With map() |  |  |  | List<String> companyNames = Arrays.asList(  "paypal", "oracle", "", "microsoft", "", "apple");  Optional<List<String>> listOptional = Optional.of(companyNames);  int size = listOptional  .map(List::size)  .orElse(0); |
| Transforming Value With flatMap() |  |  |  | Person person = new Person("john", 26);  Optional<Person> personOptional = Optional.of(person);  Optional<Optional<String>> nameOptionalWrapper  = personOptional.map(Person::getName);  Optional<String> nameOptional  = nameOptionalWrapper.orElseThrow(IllegalArgumentException::new);  String name1 = nameOptional.orElse("");  assertEquals("john", name1);  String name = personOptional  .flatMap(Person::getName)  .orElse(""); |
| Chaining Optionals in Java 8: |  |  |  | Optional<String> found = Stream.of(getEmpty(), getHello(), getBye())  .filter(Optional::isPresent)  .map(Optional::get)  .findFirst(); |

**Difference Between orElse and orElseGet():**

* To a lot of programmers who are new to Optional or Java 8, the difference between orElse() and orElseGet() is not clear.
* As a matter of fact, these two methods give the impression that they overlap each other in functionality.
* However, there’s a subtle but very important difference between the two that can affect the performance of our code drastically if not well understood.
* Let’s create a method called getMyDefault() in the test class, which takes no arguments and returns a default value:

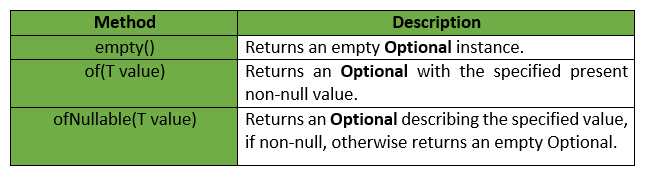
|  |
| --- |
| public String getMyDefault() {  System.out.println("Getting Default Value");  return "Default Value";  } |

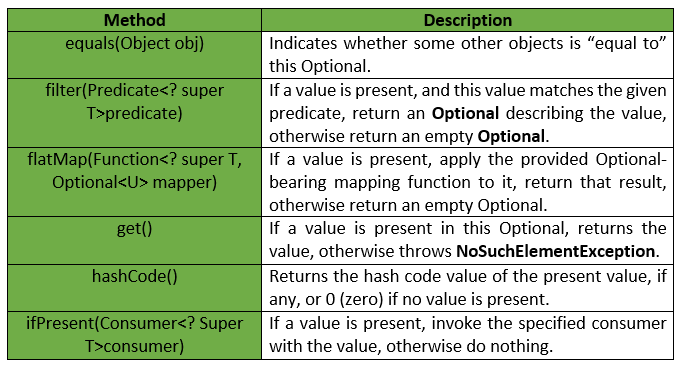
Let’s see two tests and observe their side effects to establish both where orElse() and orElseGet() overlap and where they differ:

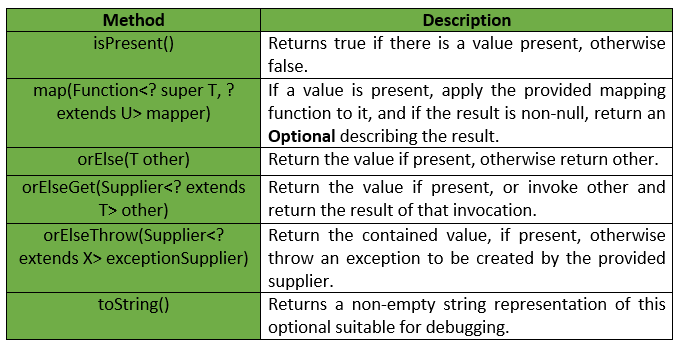
|  |
| --- |
| @Test  public void whenOrElseGetAndOrElseOverlap\_thenCorrect() {  String text = null;  String defaultText = Optional.ofNullable(text).orElseGet(this::getMyDefault);  assertEquals("Default Value", defaultText);  defaultText = Optional.ofNullable(text).orElse(getMyDefault());  assertEquals("Default Value", defaultText);  } |

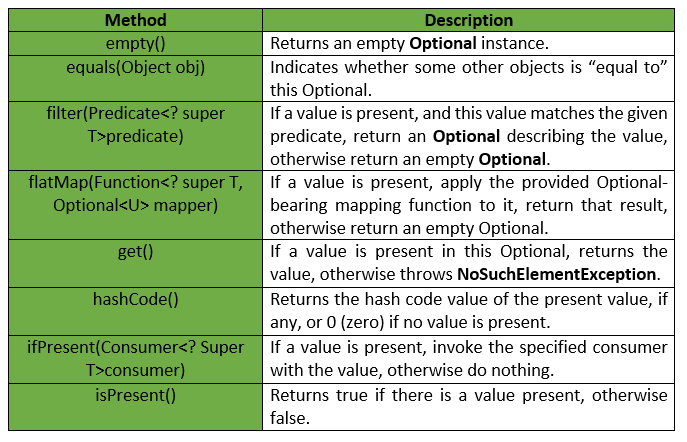
|  |
| --- |
| @Test  public void whenOrElseGetAndOrElseDiffer\_thenCorrect() {  String text = "Text present";  System.out.println("Using orElseGet:");  String defaultText  = Optional.ofNullable(text).orElseGet(this::getMyDefault);  assertEquals("Text present", defaultText);  System.out.println("Using orElse:");  defaultText = Optional.ofNullable(text).orElse(getMyDefault());//event though value exist still orElse() method will get execute once.  assertEquals("Text present", defaultText);  } |

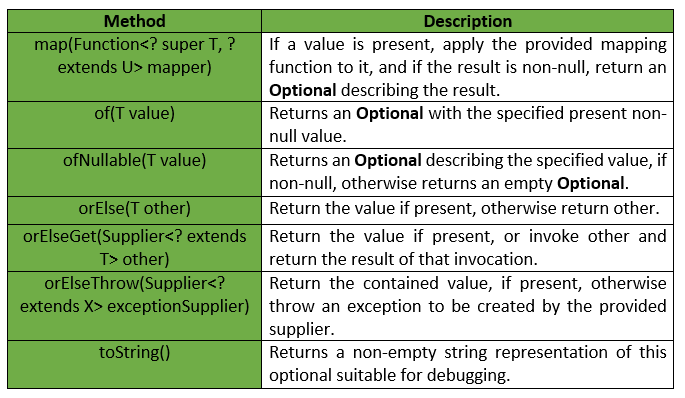
* Notice that when using orElseGet() to retrieve the wrapped value, the getMyDefault() method is not even invoked since the contained value is present.
* However, when using orElse(), whether the wrapped value is present or not, the default object is created. So in this case, we have just created one redundant object that is never used.
* In this simple example, there is no significant cost to creating a default object, as the JVM knows how to deal with such. However, when a method such as getMyDefault() has to make a web service call or even query a database, the cost becomes very obvious.











**Collectors Methods:**

Collectors is a final class that extends the Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc. Java Collectors class provides various methods to deal with elements.

**Source for below collectors 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)); |

|  |  |  |  |
| --- | --- | --- | --- |
| **S No** | **Method name** | **Method description/situation** | **Example Program** |
| 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<K, D>> groupingBy(Function<? super T, ? extends K> classifier,  Collector<? super T, A, D> downstream) |  | Map<String, Long> groupByDept1 = list.stream().collect(Collectors.groupingBy(Employee::getDeptName, Collectors.counting()));  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) |  | Map<Boolean, Set<Integer>> partitionedByEvenNumbers = integersList.stream()  .collect(Collectors.partitioningBy(i -> i%2==0); |
| 8 | partitionBy(p1,p2) |  | List<Integer> integersList = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15, 12,2,5,3,7,0);  Map<Boolean, Set<Integer>> partitionedByEvenNumbers = integersList.stream()  .collect(Collectors.partitioningBy(i -> i%2==0, Collectors.toSet()));  Even numbers:  2  4  6  8  10  12  14  Odd numbers:  1  3  5  7  9  11  13 |
| 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 | Creating list: toList() | It is used to accumulate elements into a list. It will create a new list (It will not change the current list). | List<Integer> integers = Arrays.asList(1,2,3,4,5,6,6);  integers.stream().map(x -> x\*x).collect(Collectors.toList());  // output: [1,4,9,16,25,36,36] |
| 13 | Creating set: toSet() | It is used to accumulate elements into a set, It will remove all the duplicate entries. | List<Integer> integers = Arrays.asList(1,2,3,4,5,6,6);  integers.stream().map(x -> x\*x).collect(Collectors.toSet());  // output: [1,4,9,16,25,36] |
| 14 | Creating specific collection: toCollection() | We can accumulate data in any specific collection as well | List<Integer> integers = Arrays.asList(1,2,3,4,5,6,6);  integers  .stream()  .filter(x -> x >2)  .collect(Collectors.toCollection(LinkedList::new));  // output: [3,4,5,6,6]  Here we are accumulating elements in a linked list. |
| 15 | Counting elements: Counting() |  |  |
| 16 | Collectors.summingInt | If we need to print the sum of values of one of the property from list of objects(eg: Customer/Student) while grouping by that we can use as mapper function to prepare value for key. | Map<Integer, Integer> summingBy = studentList.stream()  .filter(Objects::nonNull)  .collect(Collectors.groupingBy(Customer::getId,  **Collectors.summingInt(Customer::getAge)**)); |
| 17 | Map.Entry.comparingByKey | If you need to sort the map using keys, then we have predefined static method from Map.Entry class that comparingByKey(Comparator) it will take comparator instance and it will sort based on key of each entry it will apply comparator logic for existing value and new value(o1 and o2) | Map sortBykey = sortBy.entrySet().stream()  .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))  .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,  (oldValue, newValue) -> oldValue, LinkedHashMap::new));  System.out.println("sortBykeyIndes" + sortBykey); |
| 18 | Map.Entry.comparingByValue | If you need to sort the map using keys, then we have predefined static method from Map.Entry class that comparingByValue(Comparator) it will take comparator instance and it will sort based on key of each entry it will apply comparator logic for existing value and new value(o1 and o2) | Map sortByValue= sortBy.entrySet().stream()  .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))  .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,  (oldValue, newValue) -> oldValue, LinkedHashMap::new));  System.out.println(“sortByValue” + sortByValue); |

**Bi(Two input argument) Functional Interfaces :**

What is the purpose of the bi functional interfaces in java8?

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference. Represents a function that accepts two arguments and produces a result. This is the two-arity specialization of Function.

|  |  |  |  |
| --- | --- | --- | --- |
| Functional Interface | Abstract Method | Used Methods | Example |
| BiPredicate<T, U> | boolean test(T t, U u); |  |  |
| BiConsumer<T, U> | void accept(T t, U u); |  |  |
| BiFunction<T, U, R> | R apply(T t, U u); |  |  |
| BinaryOperator<T> extends BiFunction<T,T,T> | No abstract methods but we have two static methods  public static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)  public static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator) |  |  |
| UnaryOperator<T> extends Function<T, T> | No abstract methods but we have one static methods  static <T> UnaryOperator<T> identity() {  return t -> t;  } |  |  |

**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)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Random.html))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](https://www.linkedin.com/pulse/java-memory-permgen-vs-metaspace-incus-data-pty-ltd-drxbe/)

**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.
* To make type safety in collections, they have introduced generic.

**Record in Java 14:**

* If you’re tired of writing long and boring code in your Java programs, you’re not alone. Luckily, there’s a cool new feature called Java Records that can help you make your code shorter and easier to read.
* In this article, I’ll show you how to use Java Records and give you some examples to help you understand how they work. I’ll also explain when it’s a good idea to use Records instead of regular classes.
* Records are a better choice than classes in situations where you are primarily storing data and not defining any behaviour.

Why Records are good for storing data

* With a Record, you can define the data fields in one line of code, instead of having to define a constructor and getter/setter methods for each field in a class. This makes your code shorter, easier to read, and less prone to errors.
* Records have a built-in equals() and hashCode() method, which makes it easy to compare two instances of a Record based on their values

Data Transfer Objects (DTOs)

* Records are a good fit for DTOs, which are used to transfer data between different parts of an application. With records, you can define DTOs with just a few lines of code, reducing the amount of boilerplate code you need to write

|  |
| --- |
| public record PersonDTO(String firstName, String lastName, int age) {} |

Immutable objects

* Records are immutable by default, making them a good choice for classes that should not be modified after instantiation.
* With records, you don’t need to write any code to make the class immutable — it’s done for you automatically.

|  |
| --- |
| public record Temperature(double value, String unit) {} |

Simple value types

* Records are a good fit for classes that represent simple value types.
* For example, you might use a record to represent a point in a two-dimensional space.

|  |
| --- |
| public record Point(int x, int y) {} |

API responses

* Records are a good choice for representing responses returned by an API.
* With records, you can define a class with just the fields you need, making it easier to work with the API response.

|  |
| --- |
| public record ErrorResponse(int code, String message, String additionalInfo) {} |

Configuration settings

* Records are a good fit for classes that represent configuration settings.
* With records, you can define a class with just the fields you need, making it easier to manage the configuration settings for your application.

|  |
| --- |
| public record DbConfig(String databaseUrl, String username, String password) {} |

Do we really need Records?

I will say no but it is better. there is a way to reduce the boilerplate code with Lombok annotations.

Example:

* Person Record

|  |
| --- |
| public record Person(String name, int age) {} |

* Lombok Annotated Person class

|  |
| --- |
| @Data  @AllArgsConstructor  public class Person {  private String name;  private int age;  }  //@Data Annotation Generates getters for all fields, a useful toString method, and hashCode and equals implementations that check all non-transient fields. Will also generate setters for all non-final fields, as well as a constructor.  // Equivalent to @Getter @Setter @RequiredArgsConstructor @ToString @EqualsAndHashCod |

However, there are a few reasons why records might be considered for the above-mentioned scenariosConciseness

→ Records allow you to define a class with properties and a constructor in a single line of code. With Lombok, you need to use multiple annotations (@Data, @AllArgsConstructor) to achieve the same functionality, which can be more verbose and harder to read.

Immutability

→ Records are immutable by default, which means that their properties cannot be changed after they are created. This can help prevent bugs and improve code reliability.

* Ultimately, the choice between records and Lombok annotated classes depends on your specific needs and preferences. If you value conciseness and immutability, records might be a better choice. If you value convenience and code generation, Lombok annotated classes might be a better choice.
* Records are the latest and greatest feature in the Java language that can make your code concise, readable, and maintainable. Whether you’re dealing with simple or complex data structures, records can be your go-to solution. By comparing records to Lombok, I’ve shown you the benefits of using records in your Java programs.