**Java 8**

**Features of Java 8**

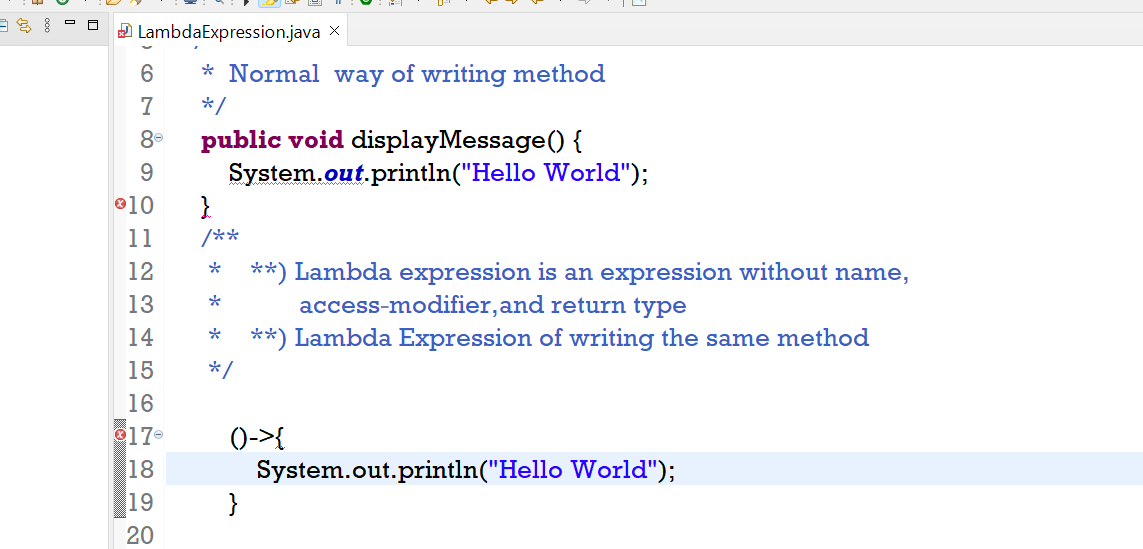
* Lambda Expression
* Functional Interface
* Default methods in Interface
* Static methods inside Interface
* Predefined Functional Interface (Predicate , Function , Supplier , Consumer)
* Method Reference and Constructor Reference **(::)**
* Stream API (Bulk Operation on Collection)
* Data And Time API **(Joda Time API)**

**Advantage**

* Simplify Programming (Easy and concise code)
* To Utilize functional programming benefits.
* To enable parallel programming(processing).

**Lambda Expression**

* To enable functional programming in Java
* Write more readable, maintainable and concise code.
* To use APIs very easily and efficiently.
* To enable parallel processing.
* It is an anonymous function . (Not having name , not having modifiers , not having return type)

****

**Functional Interface**

* An Interface which contains only single abstract method.
* It is also called SAM (Single Abstract Method).
* We can also keep default method and static method from 1.8 version onwards.
* We can keep any number of default and static method in Functional Interface
* The restriction is only for abstract method.(Only One abstract method shuld be in Functional Interface)
* We use @FunctionalInterface annotation to denote functional interface.
* This annotation is only for compiler to indicate if developer is doing any issue just let them know that the declared interface is not a functional interface.

**A screenshot of a computer code

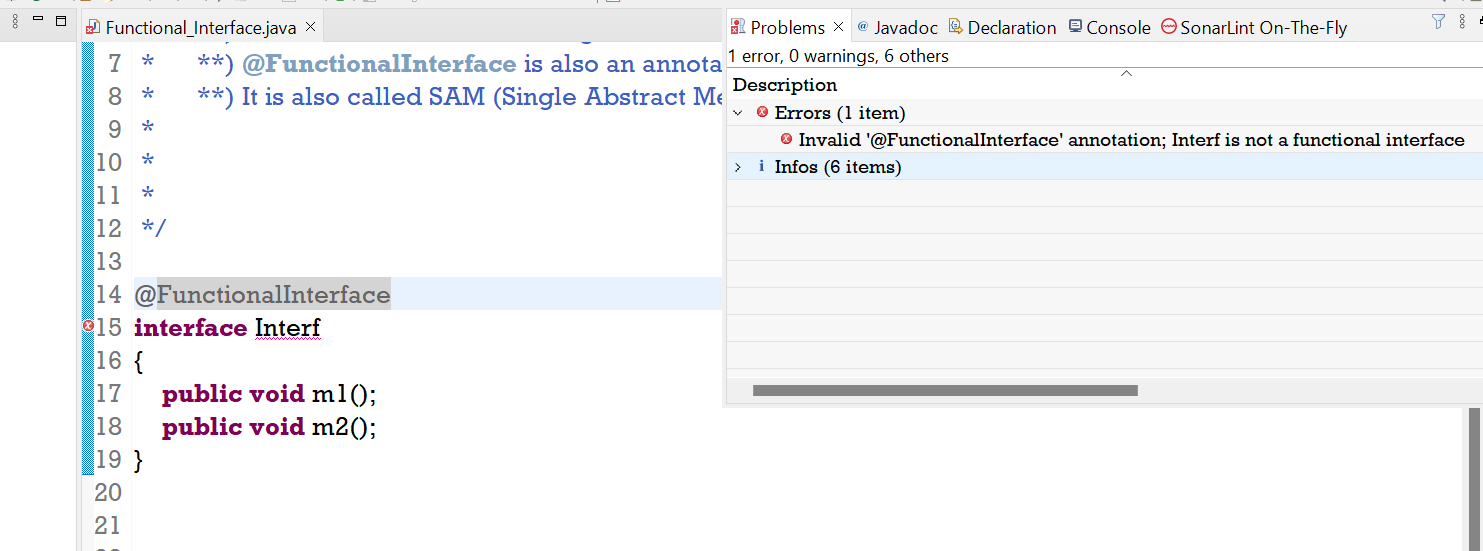
AI-generated content may be incorrect.**

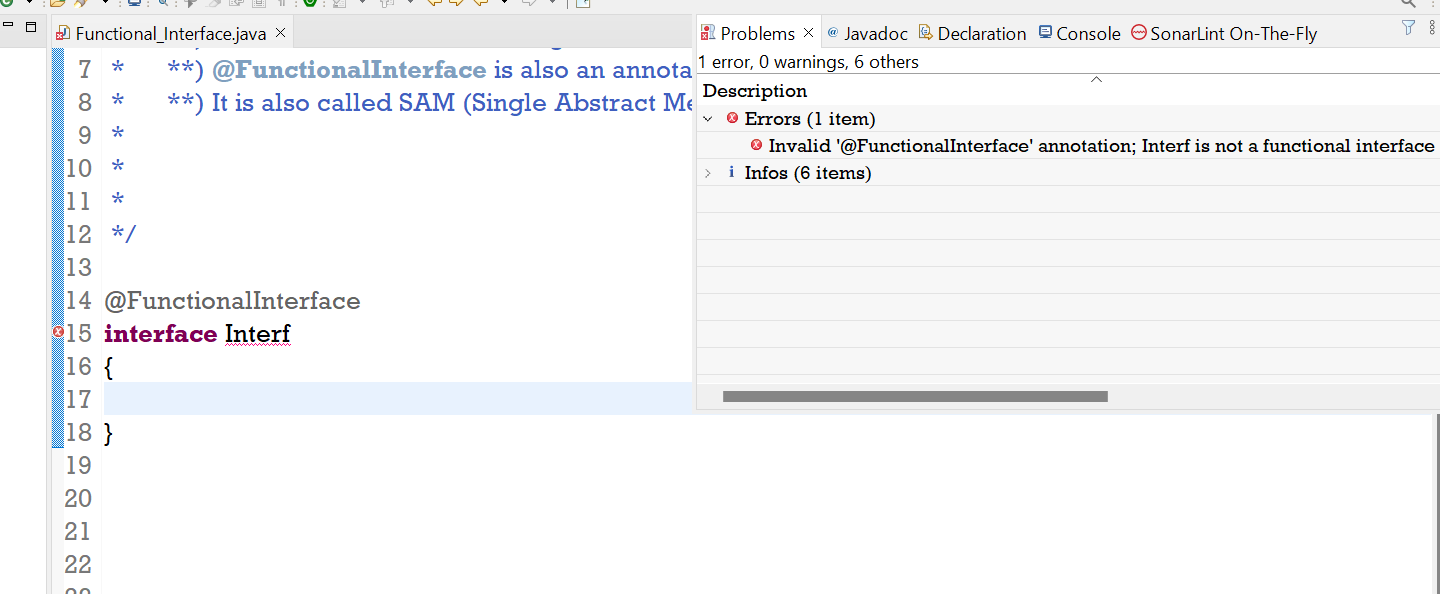
**Some of already existing Functional Interfaces**

**Runnable** 🡪 run() method

**Callable** 🡪 call() method

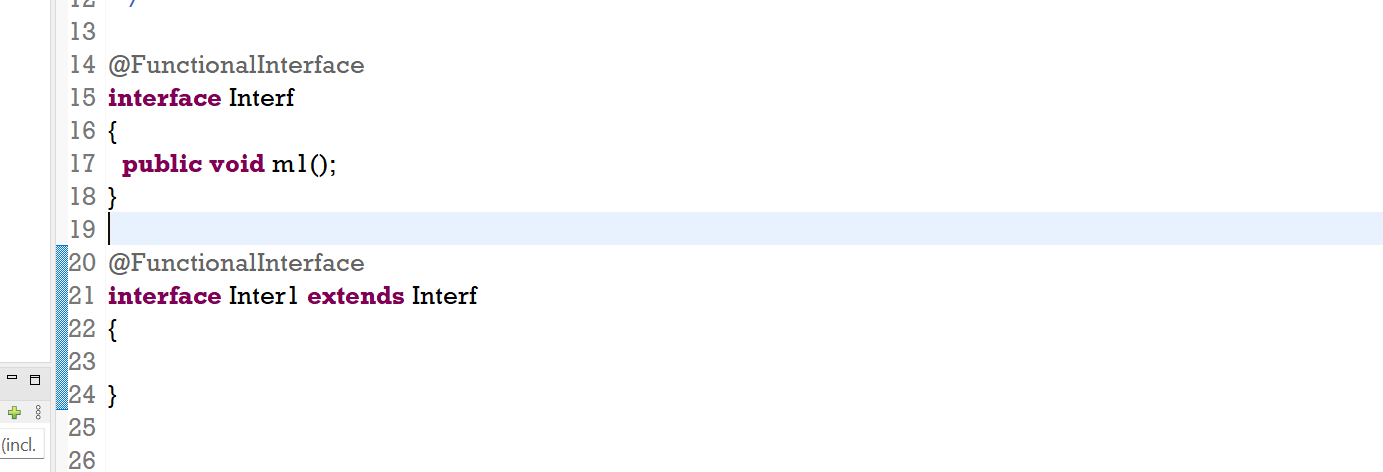
**Comparable** 🡪 compareTo method

****

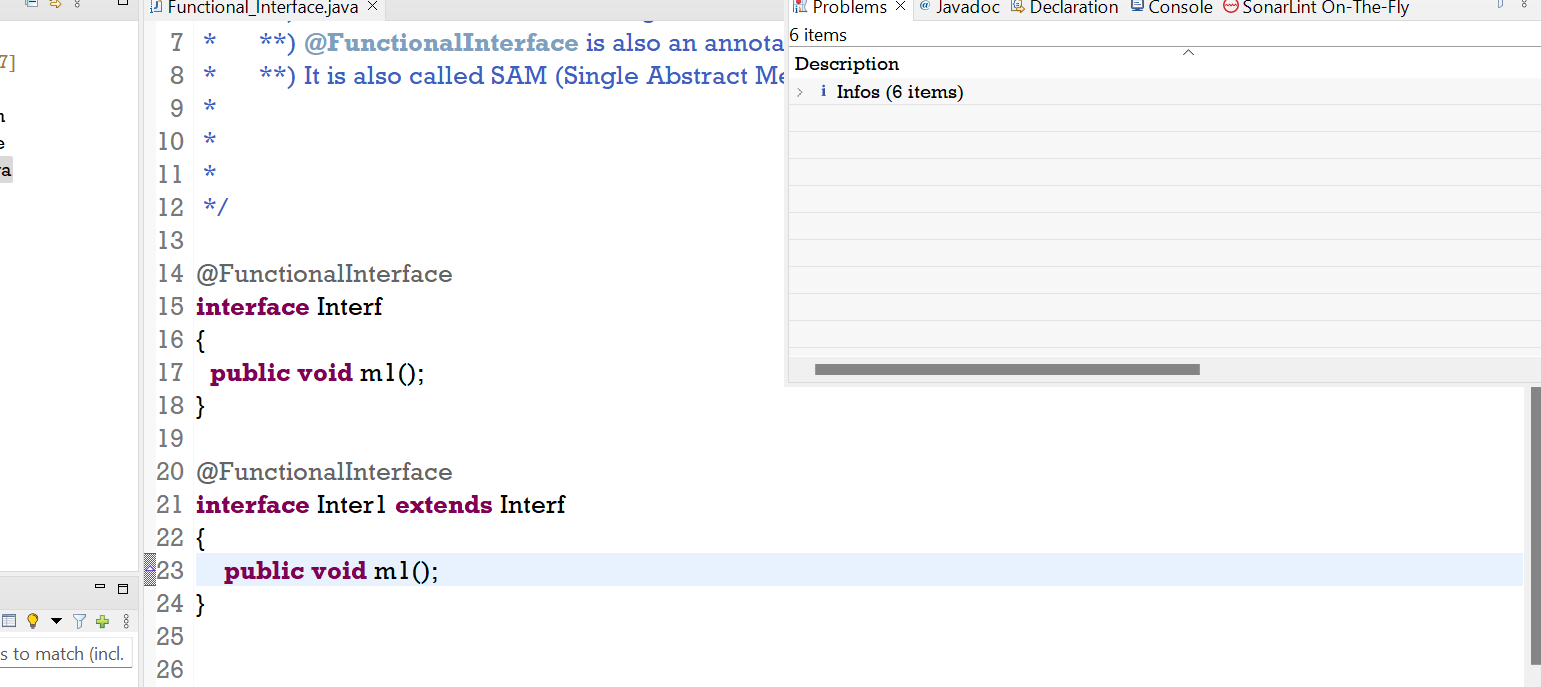
****

Above 2 Screenshot is an example of not a valid Functional Interface Example.

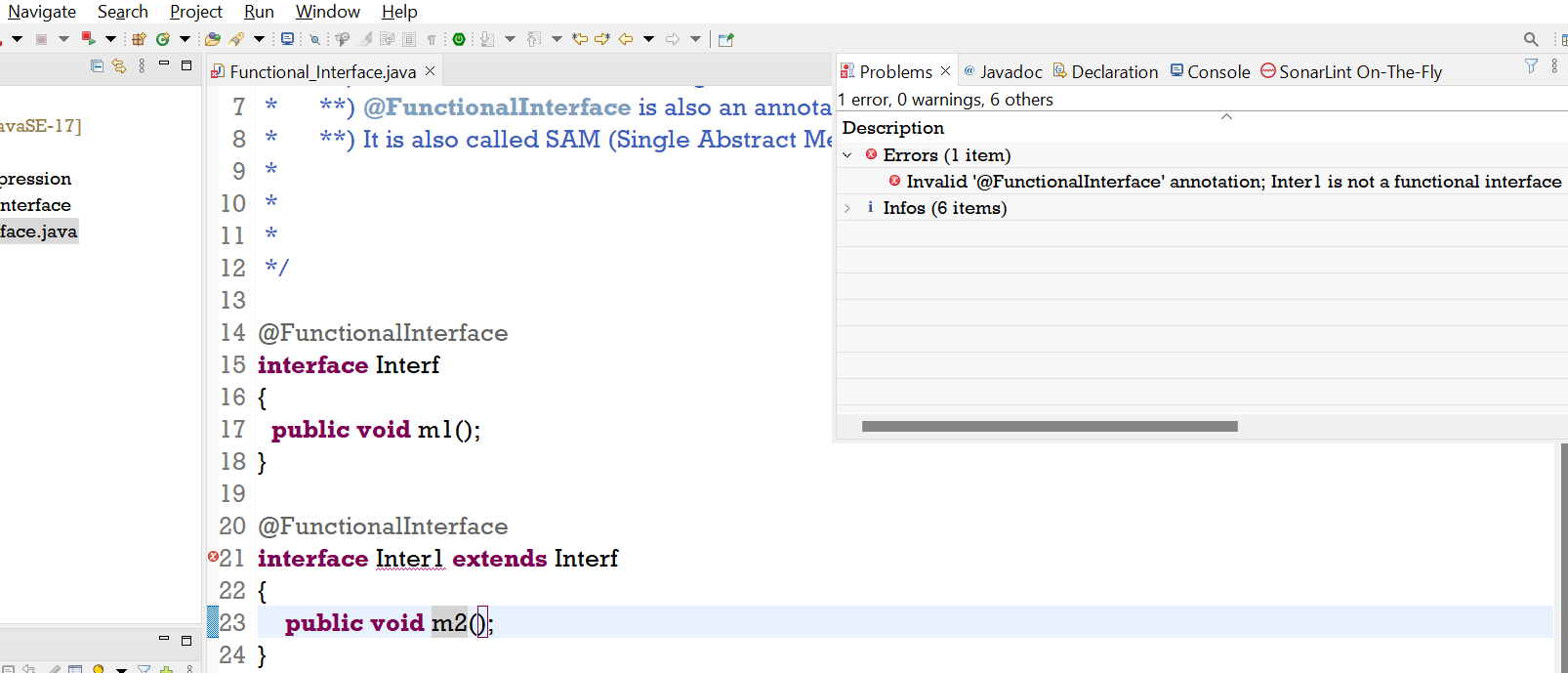
**Functional Interface W.R.T Inheritance**

****

* Child is Functional Interface and have no method but its parent have one abstact method – **Valid case**

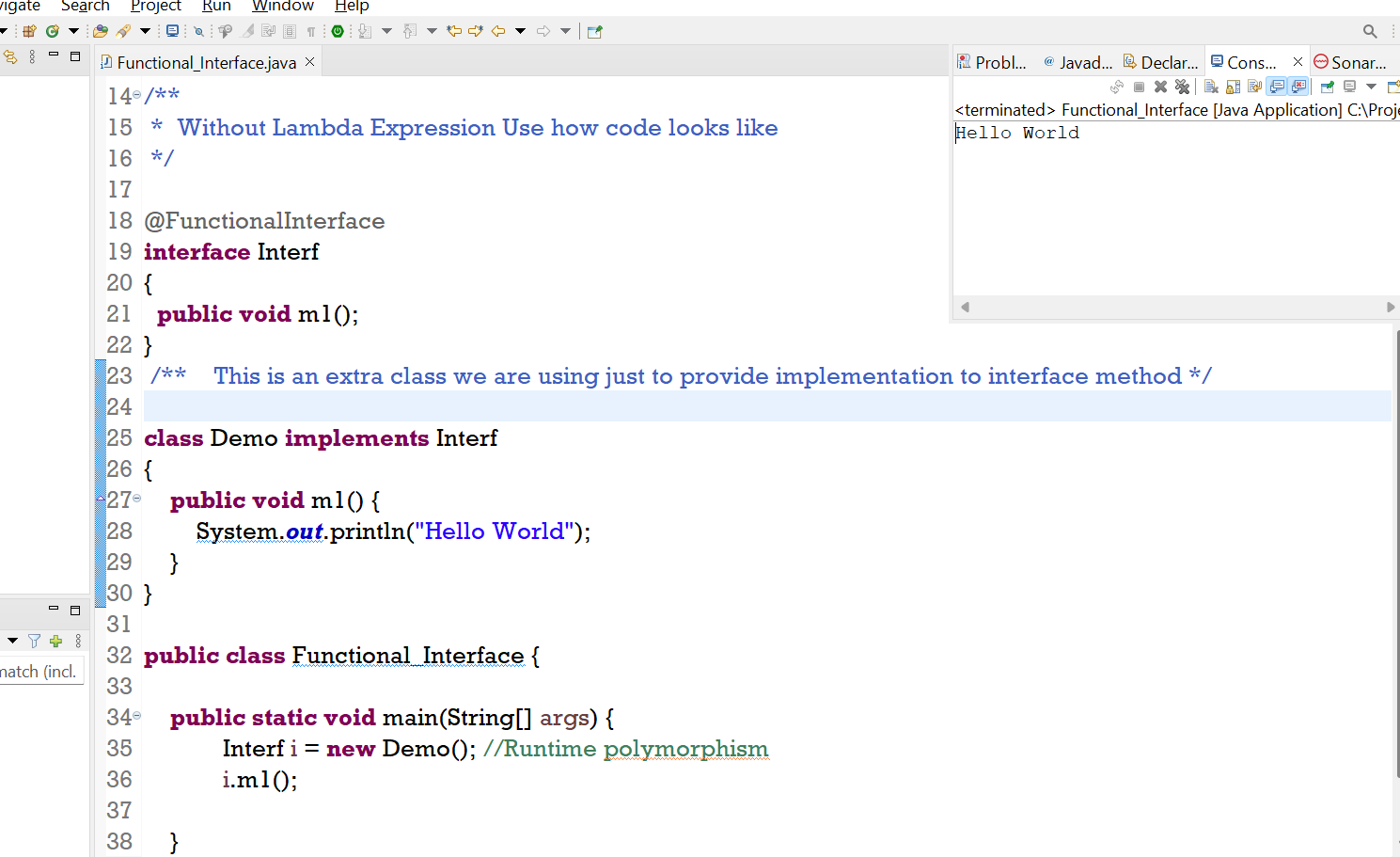


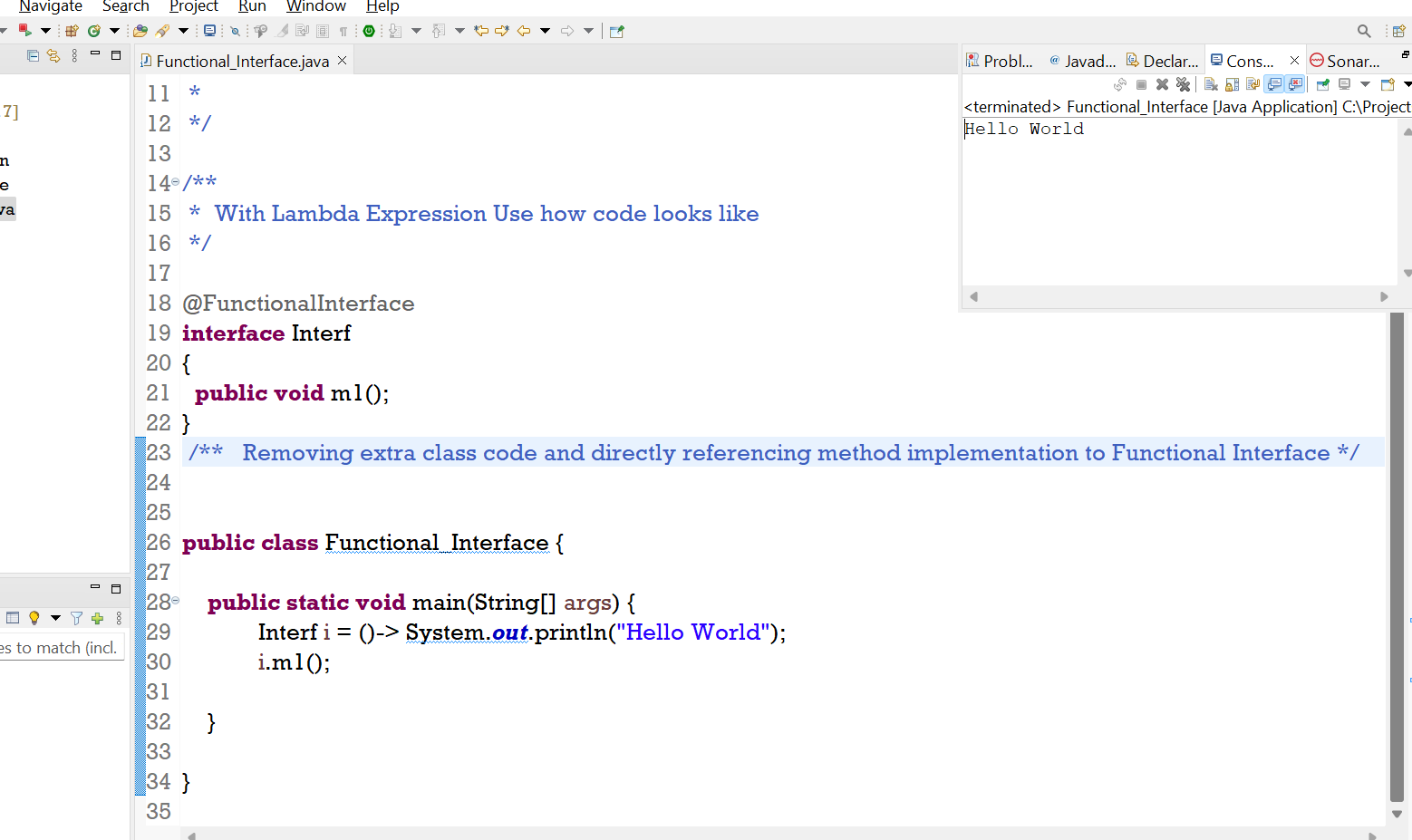
* If Child contains the same abstract method which parent contains - **Valid case**

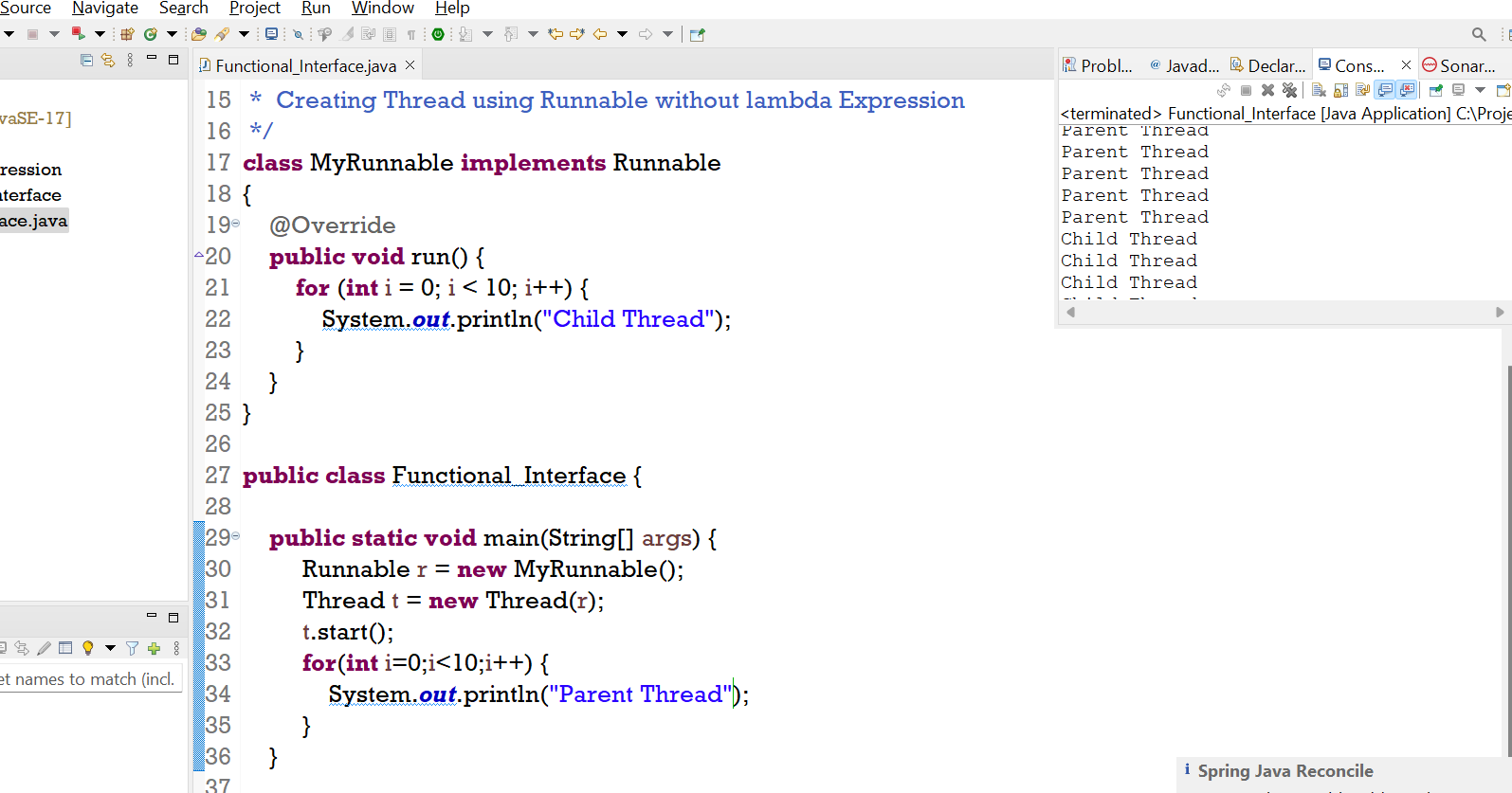


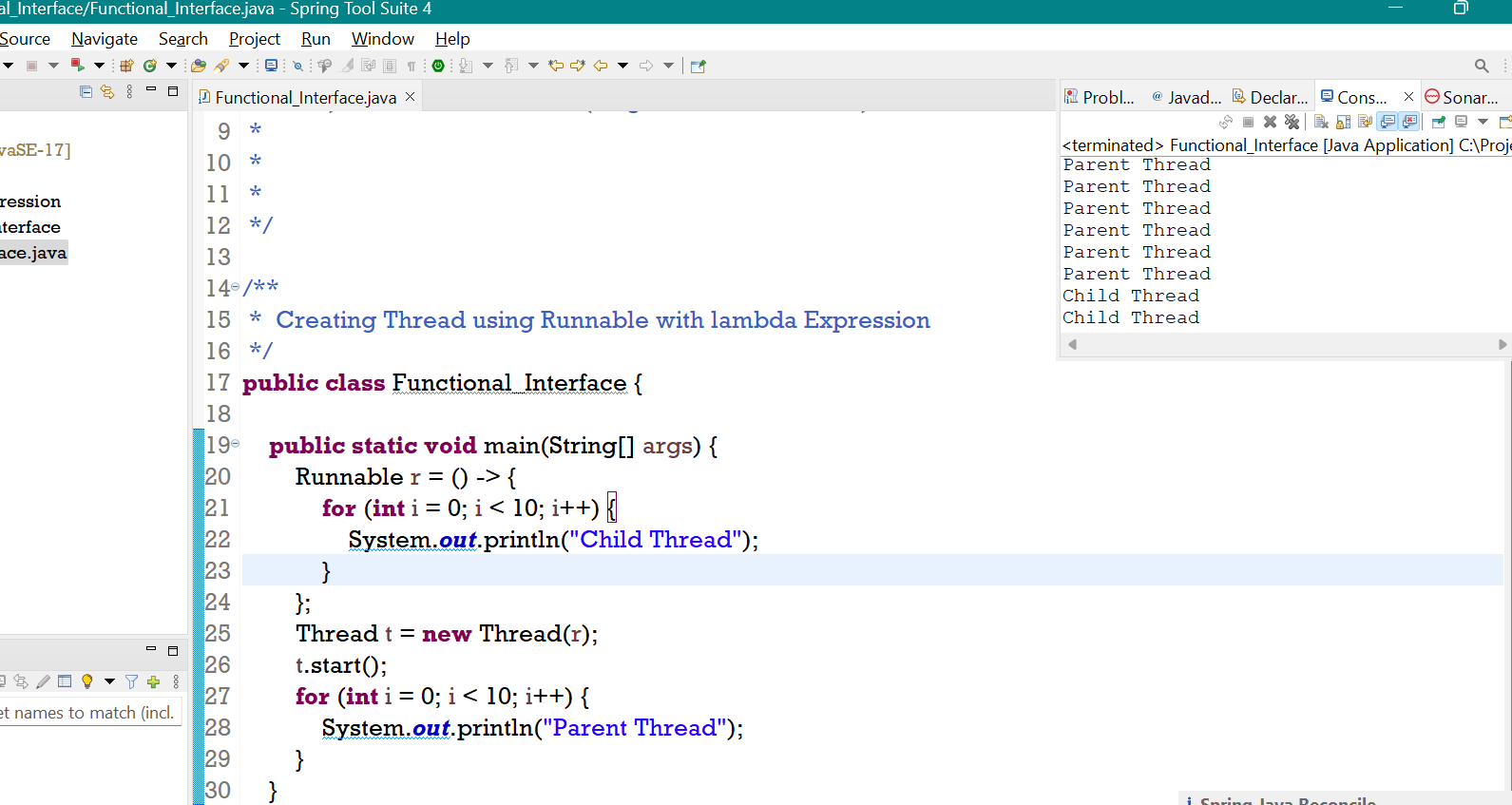
* When child have different abstract method than parent – **Invalid case of Functional Interface**

How Lambda Expression is a better approach then earlier









**Note** :

**Lambda expression provide method implementation and functional interface provide reference to hold that implementation.**

* To invoke lambda expression functional interface is required.

**Sorting of Array List without using Lambda Expression.**

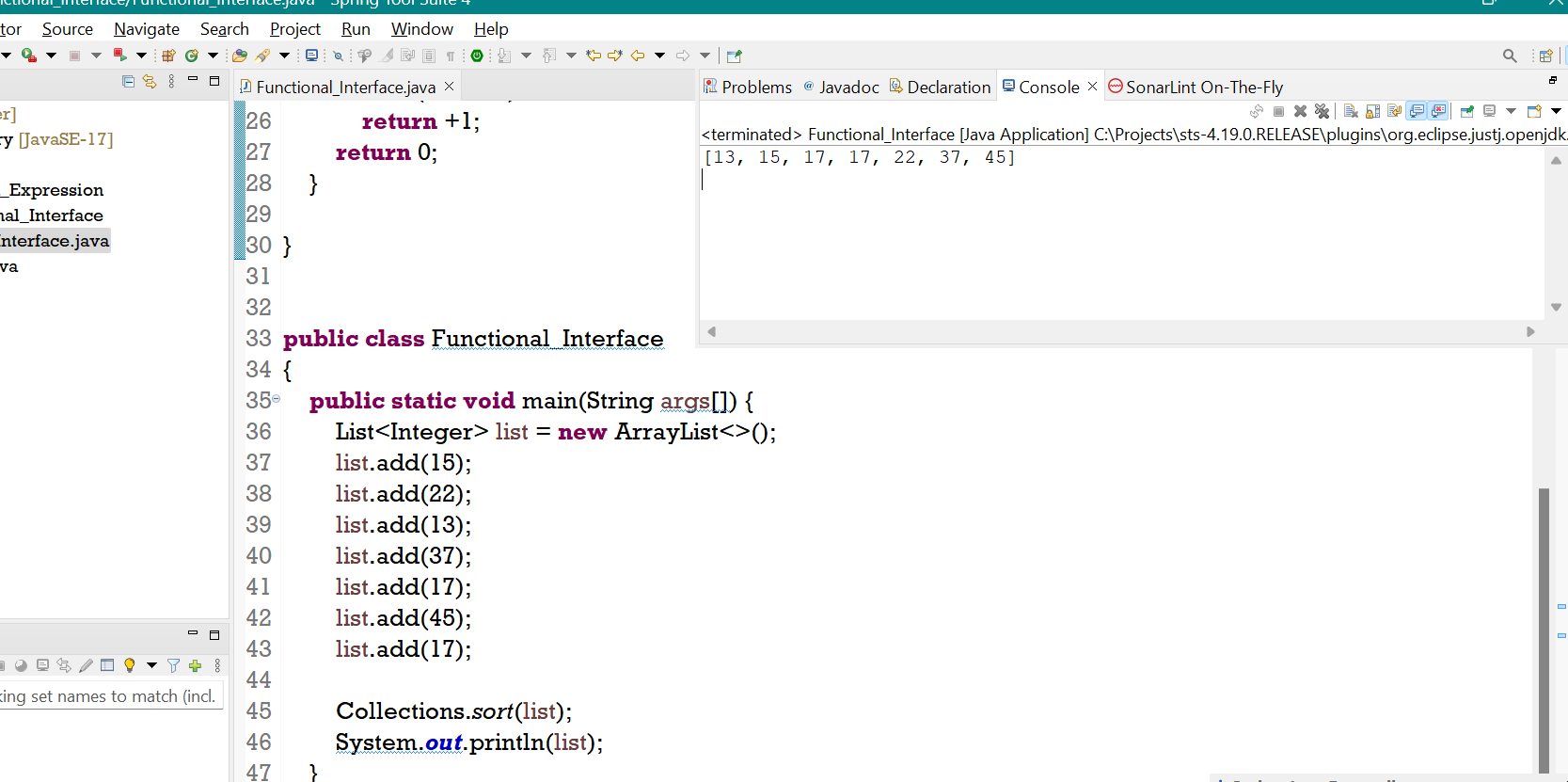
**Interface** 🡪 Comparator

**Method** 🡪 public int compare (Object obj1,Object obj2)

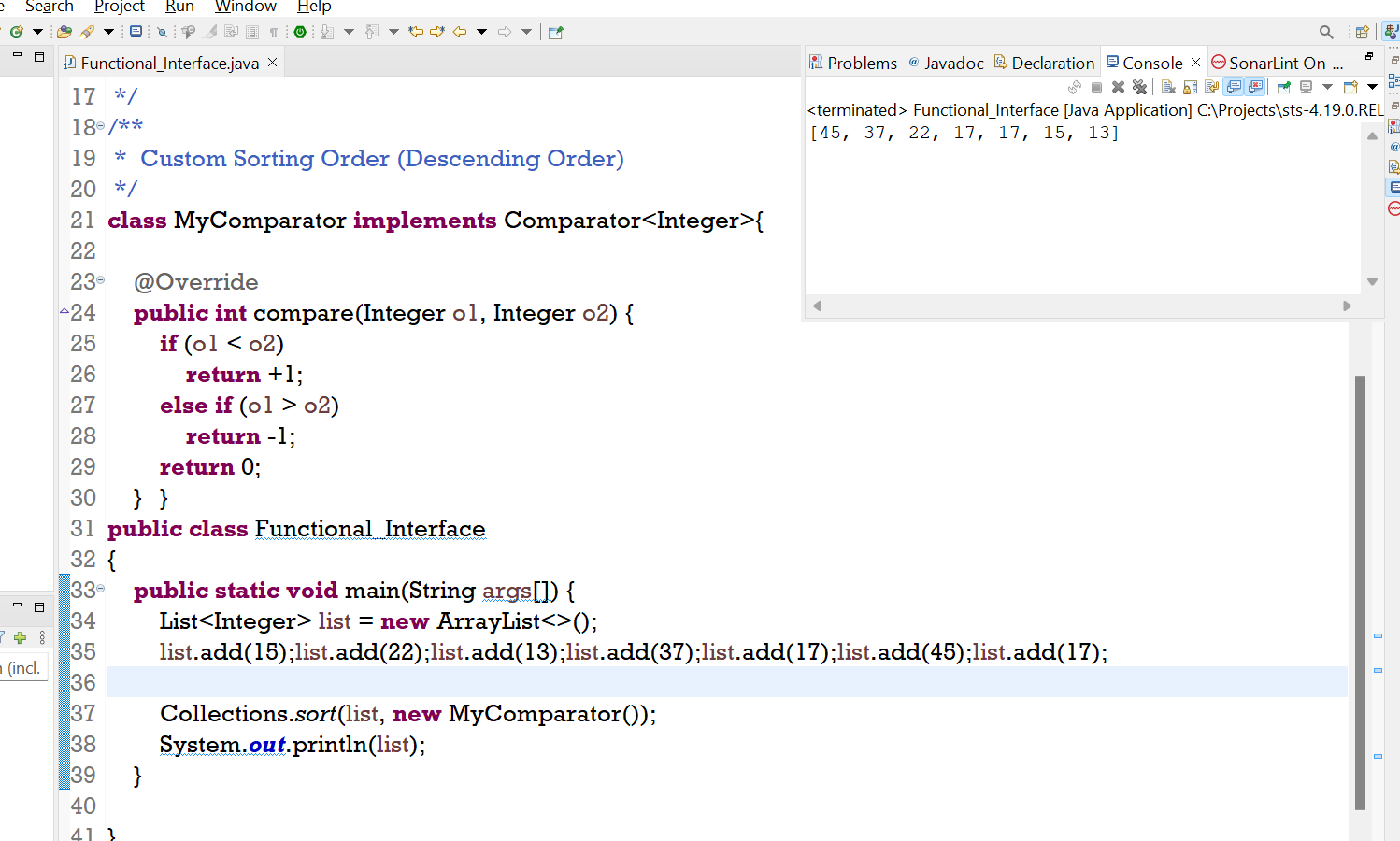
* **If Object1 has to come before Object 2 , return -ve**
* **If Object1 has to come after Object 2 , return +ve**
* **If Object1 equals Object 2 , return 0**

**List<Integer> l = new ArrayList<>();**

**Default Natural Sorting** ------------- **Collections. Sort (l) --- Default natural sorting order.**



**Custom Sorting Order**

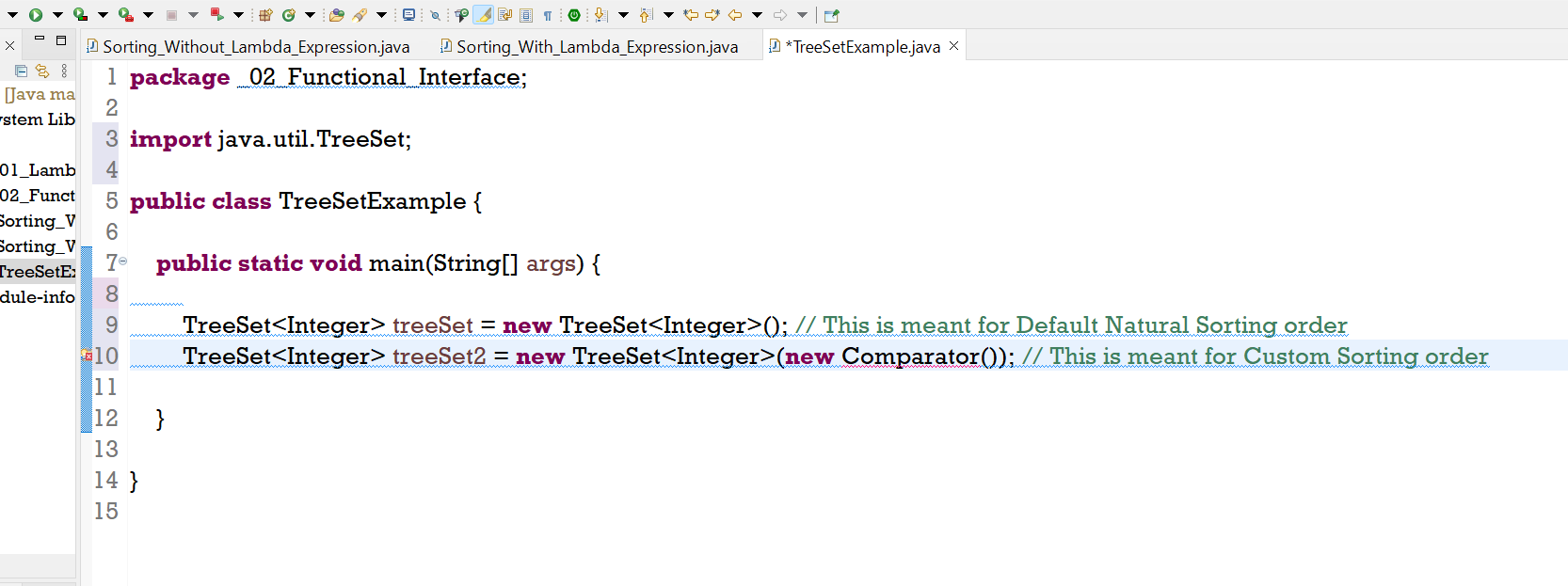
****

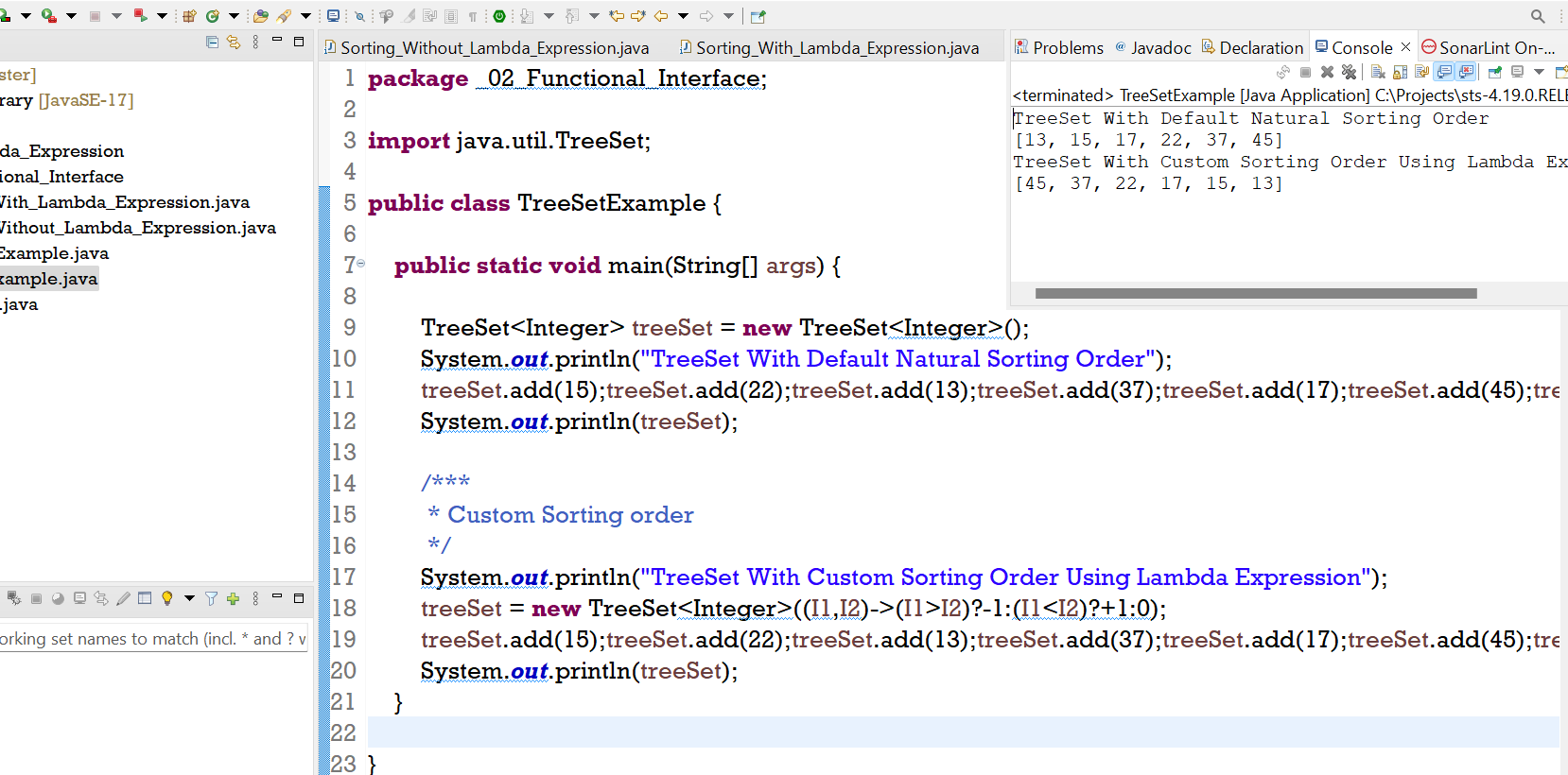
**Sorting of Array List with using Lambda Expression.**

**A screenshot of a computer

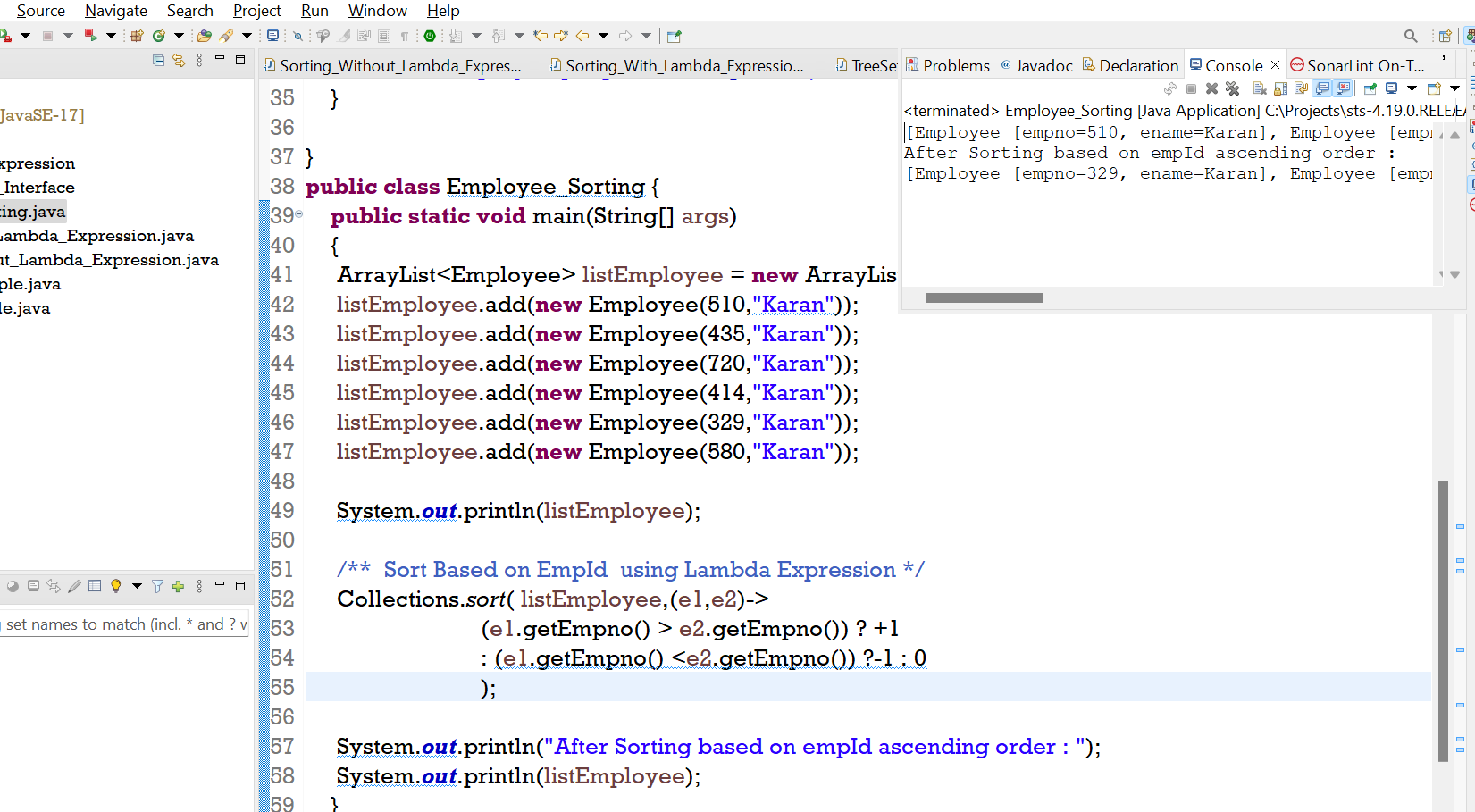
AI-generated content may be incorrect.**

**Sorting In Tree Set**

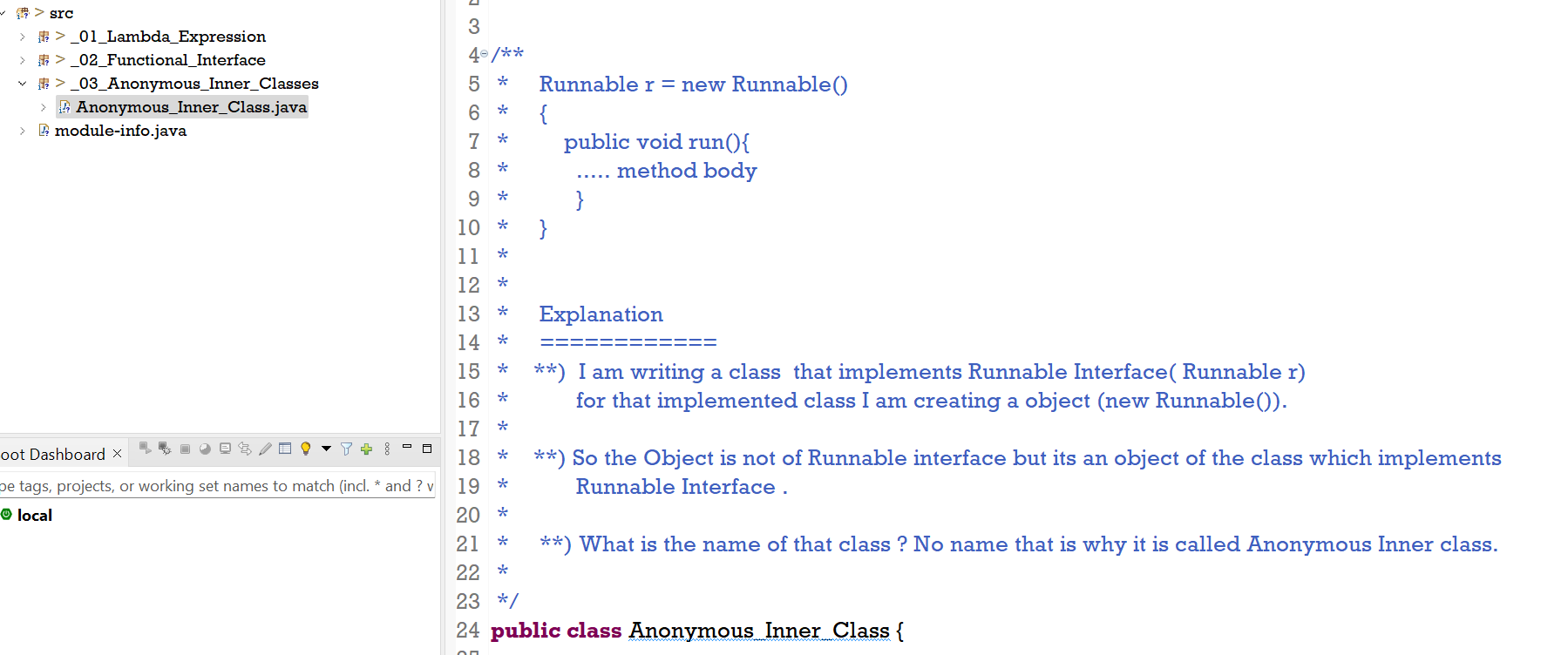


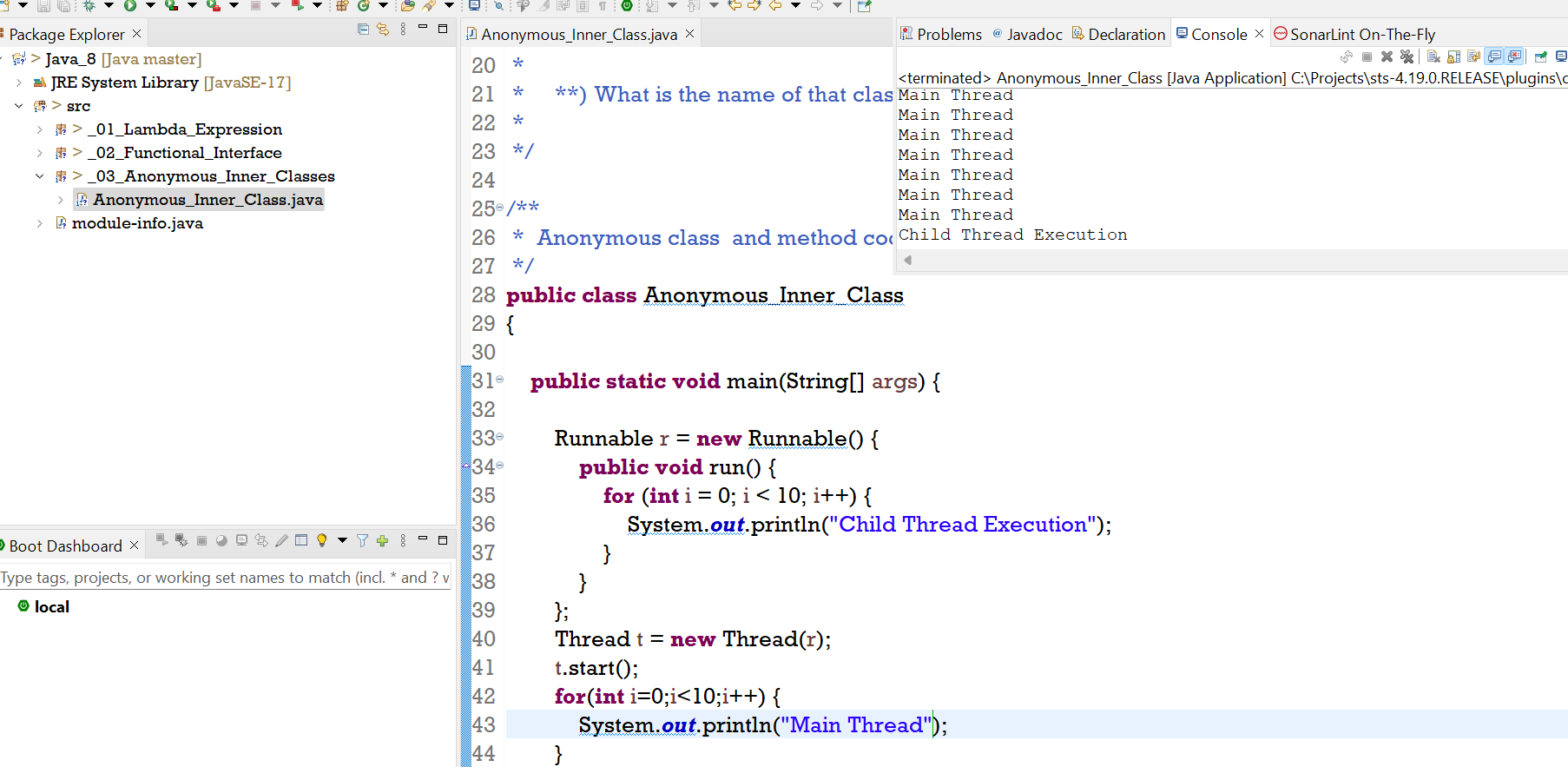


**Sorting on Custom Object (Employee object)**



**Anonymous Inner Class vs Lambda Expression**

****

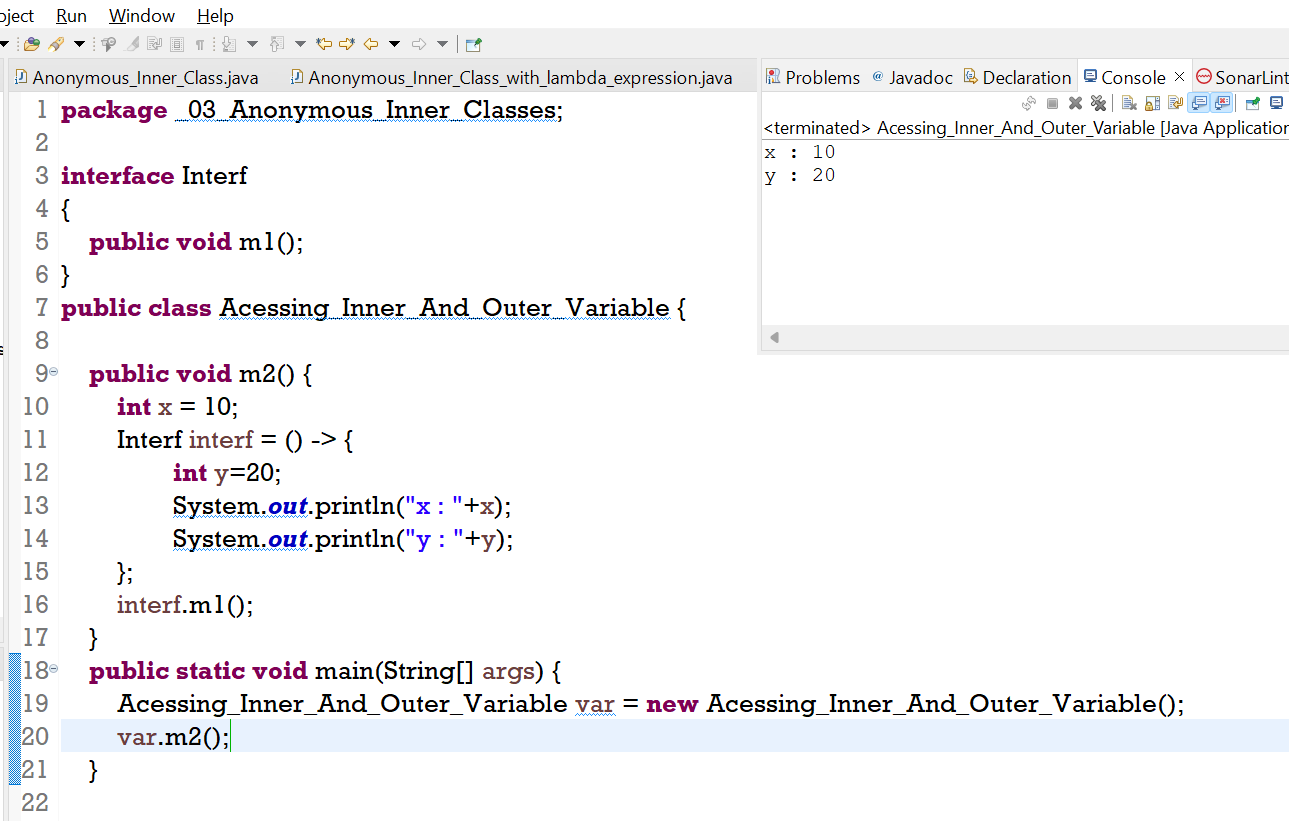
****

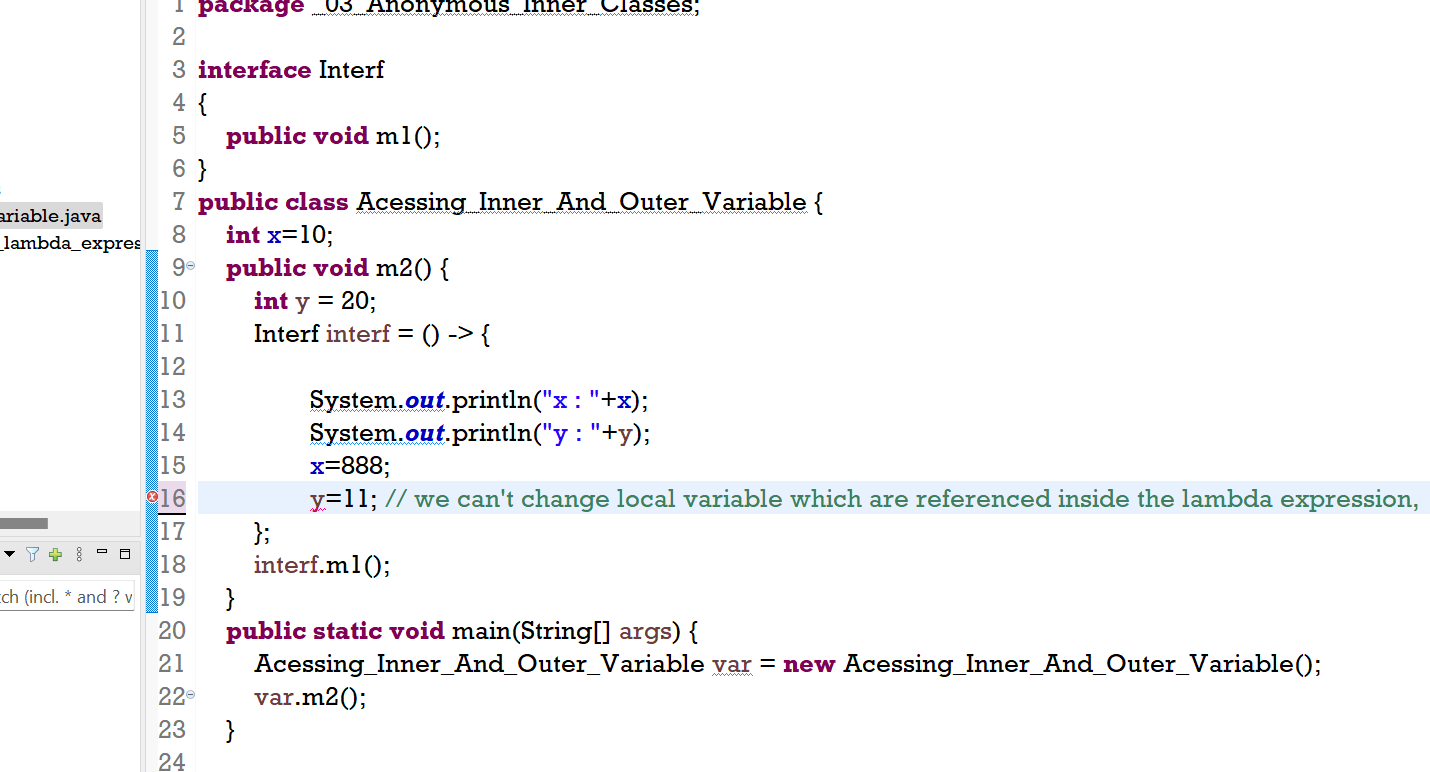
**We can replace anonymous inner class with lambda expression**

* When Anonymous Inner class extends Concrete class – No
* When Anonymous Inner class extends abstract class - No
* When Anonymous Inner class implements interface with multiple methods – No
* When Anonymous Inner class implements interface and have **single abstract method** – **Yes**

A close-up of a list of text

AI-generated content may be incorrect.





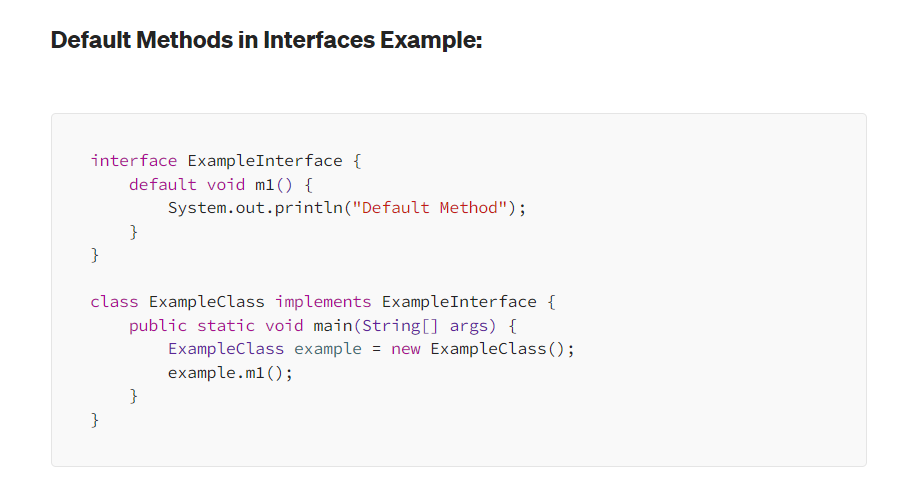
**Default Methods**

* Till Java 1.7 , we can only define **public abstract** method inside **Interface**.
* With the introduction of default method in 1.8 , It is possible to provide method implementation

to Interface using default keyword.



* Interface default method are by default available to all implementation classes.
* Based on requirements, an interface class can use these default methods directly or can override.



* These are also called **Defender methods.**
* The main advantage of default methods is we can add new functionality to interface

Without affecting the implementation classes.

Note:

* We can’t override Object class methods as default methods inside an interface; otherwise, we get a compile-time error.



A close-up of a message

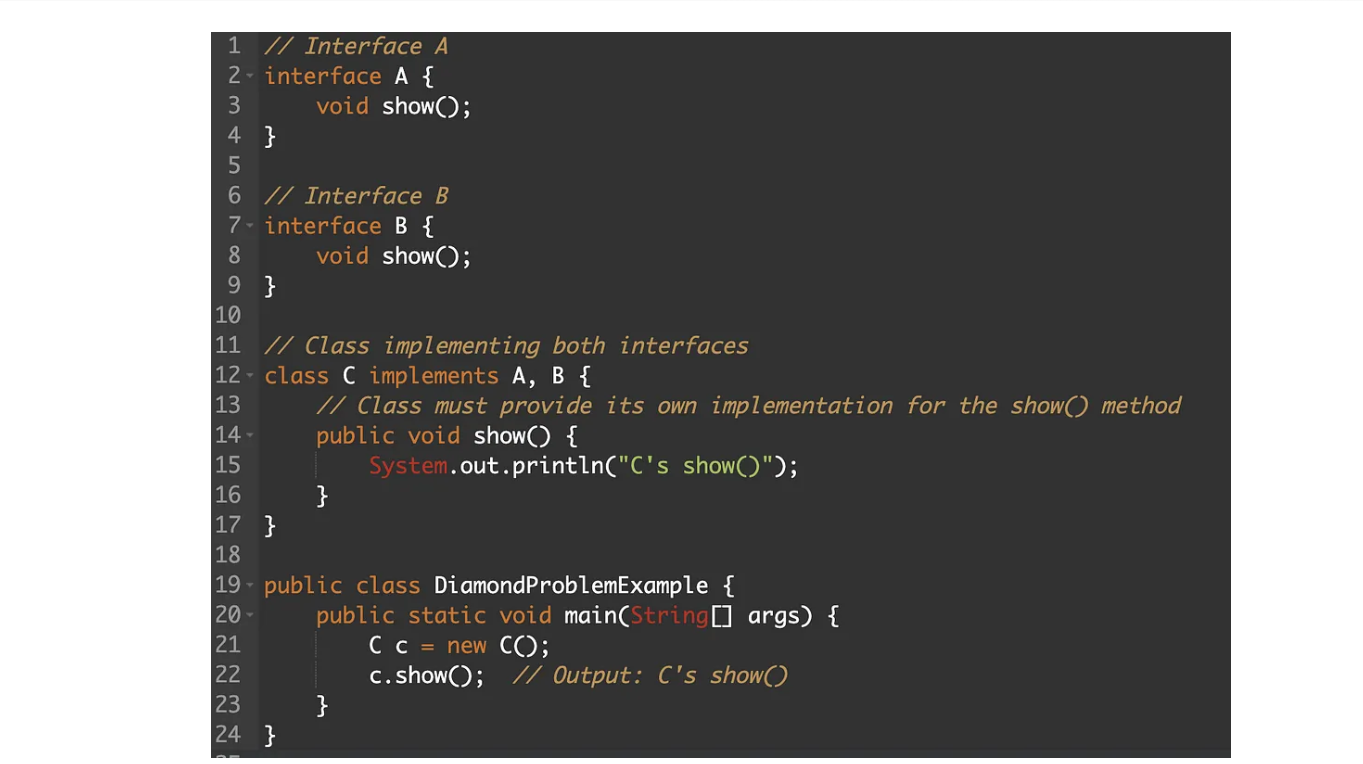
AI-generated content may be incorrect.

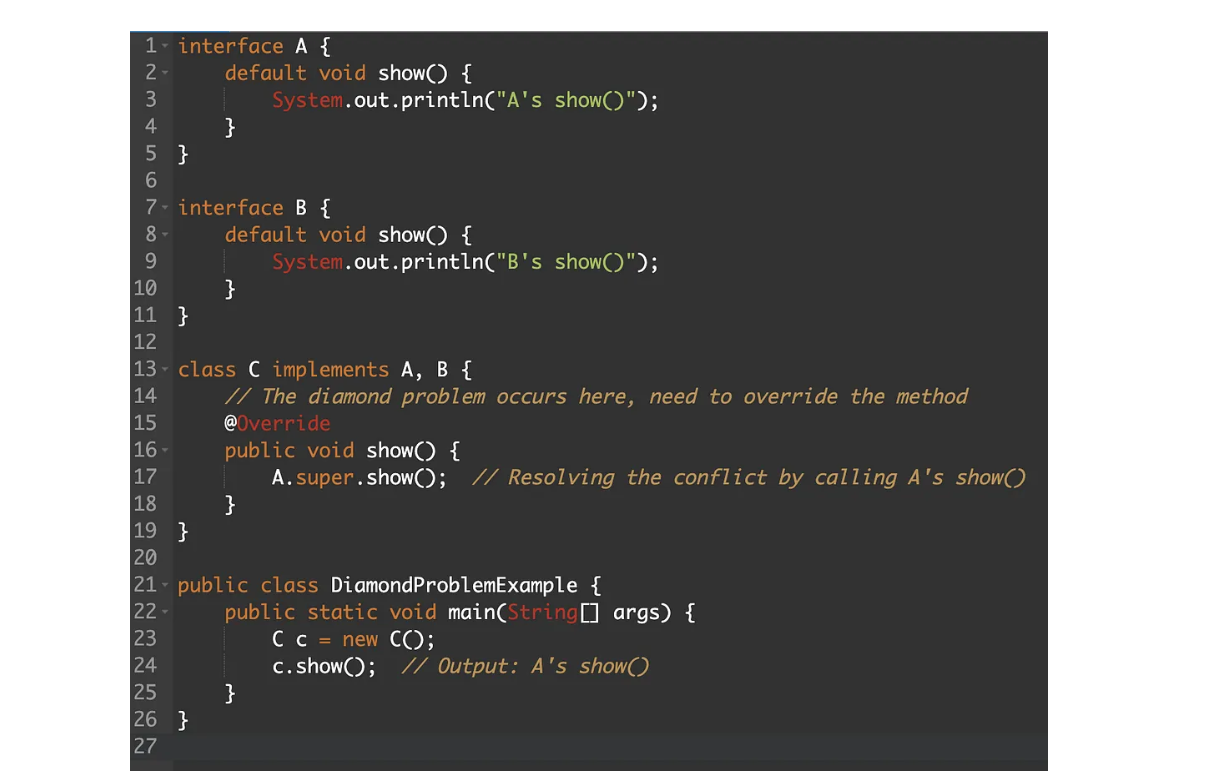
**Default method w.r.t Multiple Inheritance**

* If both interfaces contains same method then it is important to define the method

In child class where both the interfaces is implemented else it will give ambiguity.

* In order to call interface A show() method we have to call **A.super.show()** method.





**Static Methods**

* Inside interface it has been introduced in 1.8 v
* It is beneficial to define utility method that are related to interface.
* By default , interface static method is not available to its implementation class.
* Override concept is not applicable for Interface containing static methods.



A screenshot of a computer code

AI-generated content may be incorrect.

A close-up of a text

AI-generated content may be incorrect.

A screenshot of a computer program

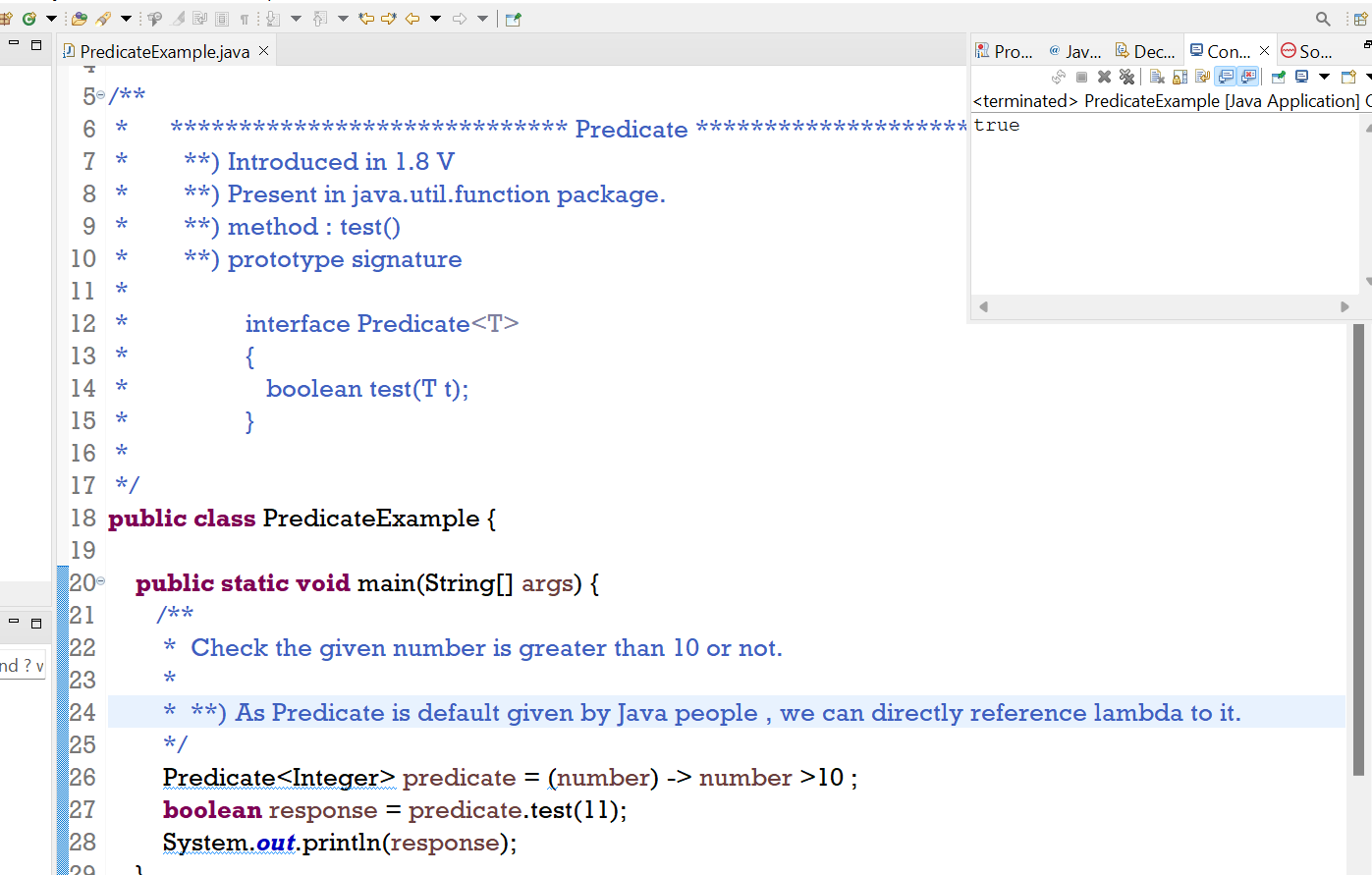
AI-generated content may be incorrect.

**Predefined Functional Interface**

* Predicate - test()
* Function - apply()
* Consumer - accept()
* Supplier – get()

Default functional interface provided by Java inside **java.util.function** interface.

**Predicate**

****

**Predicate Joining**

* Some methods help to achieve predicate join scenario.
* **Methods are :**

Let us suppose we have 2 predicate P1 and P2 then ,

* **and ()** : P1.and(P2), This will check P1 and P2 both condition will satisfy

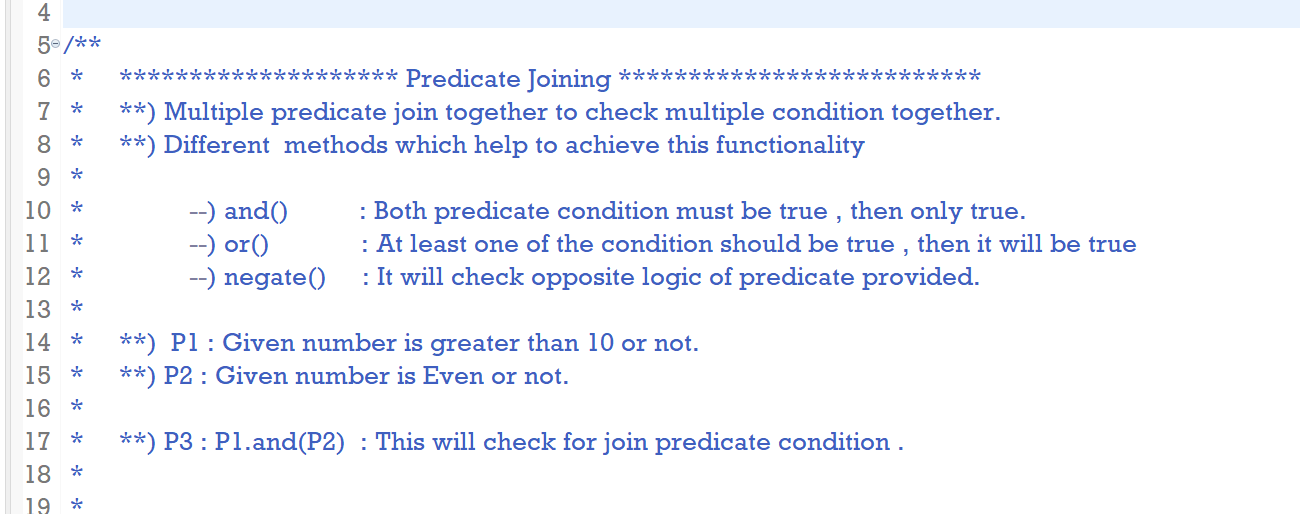
or not . If Yes, then only true else false.

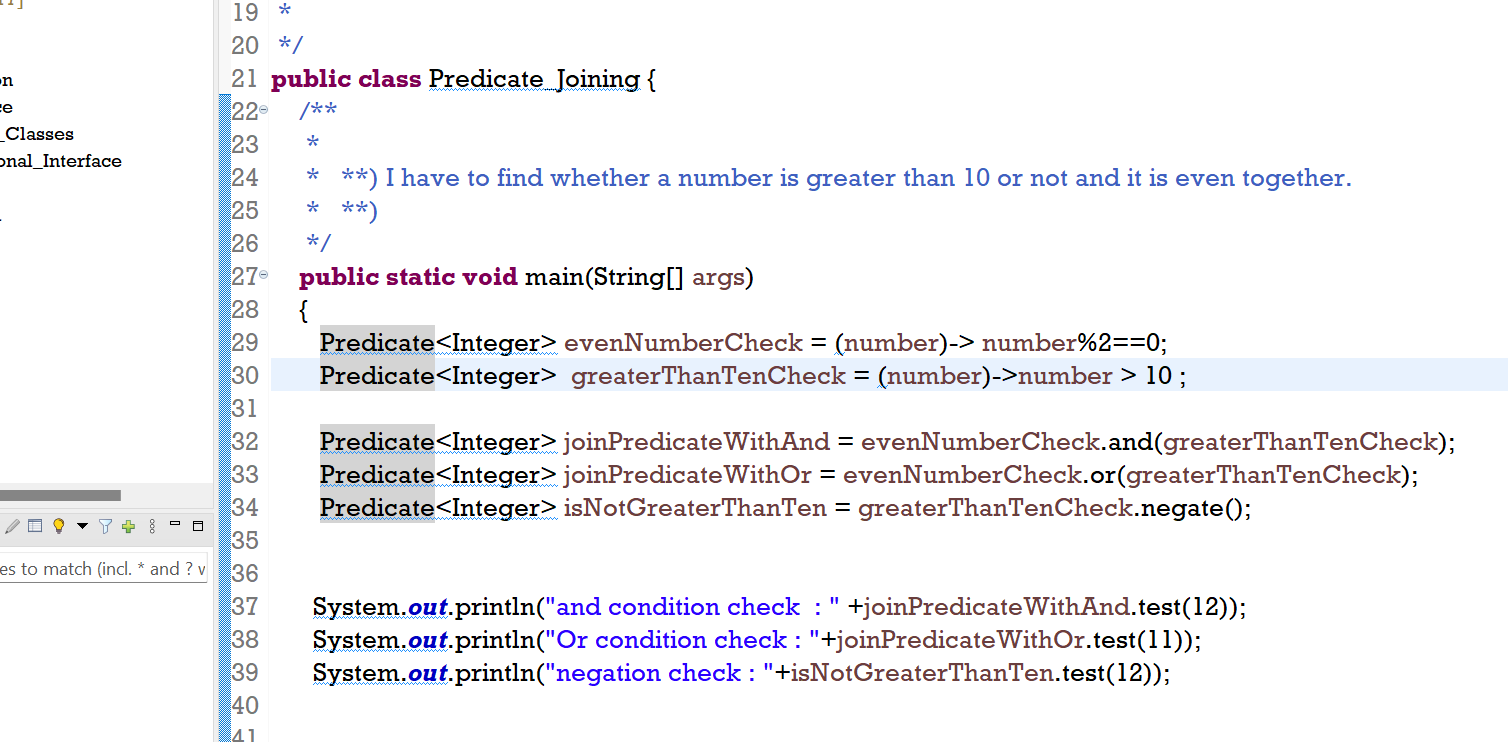
* **or()** : P1.or(P2) , This will check P1 or P2 at least one condition should satisfy

If any one satisfy return true else false.

* **negate()** : P1.negate() : It will check opposite to P1 logic If satisfied return true

else false.





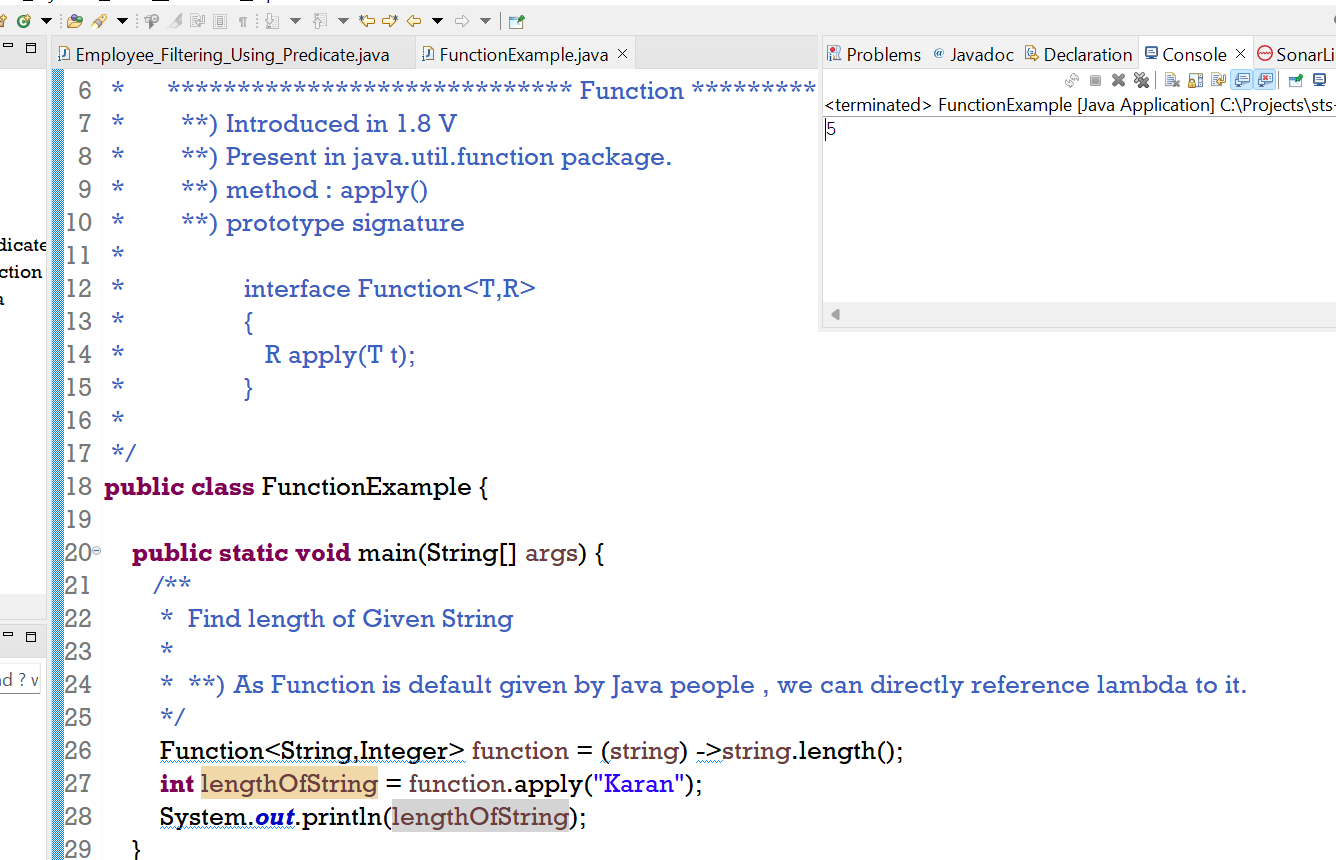
A screenshot of a computer

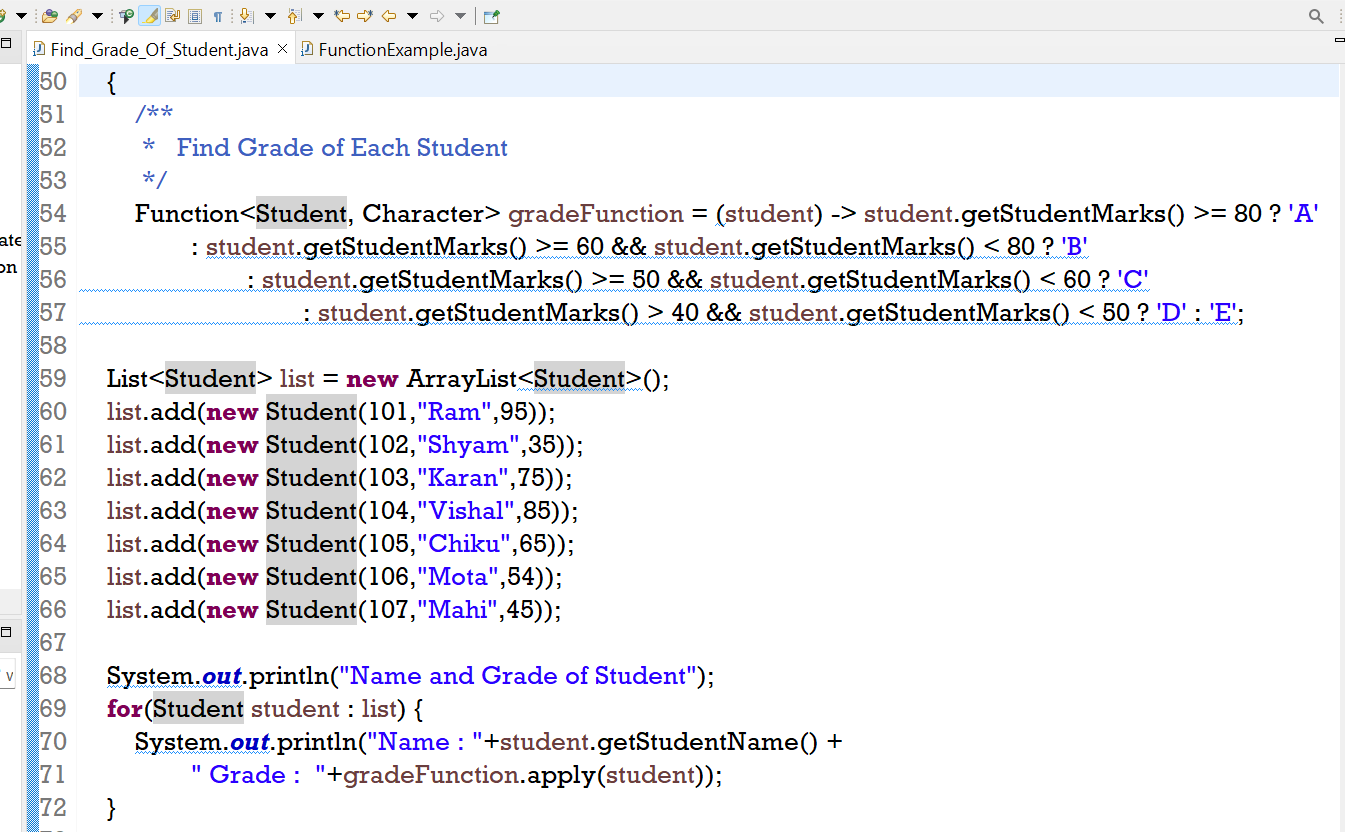
AI-generated content may be incorrect.

**Function**

* when our input type can be anything and output type can be anything then

we can go with **Function** Functional Interface.





**Function Chaining**

**Methods** :

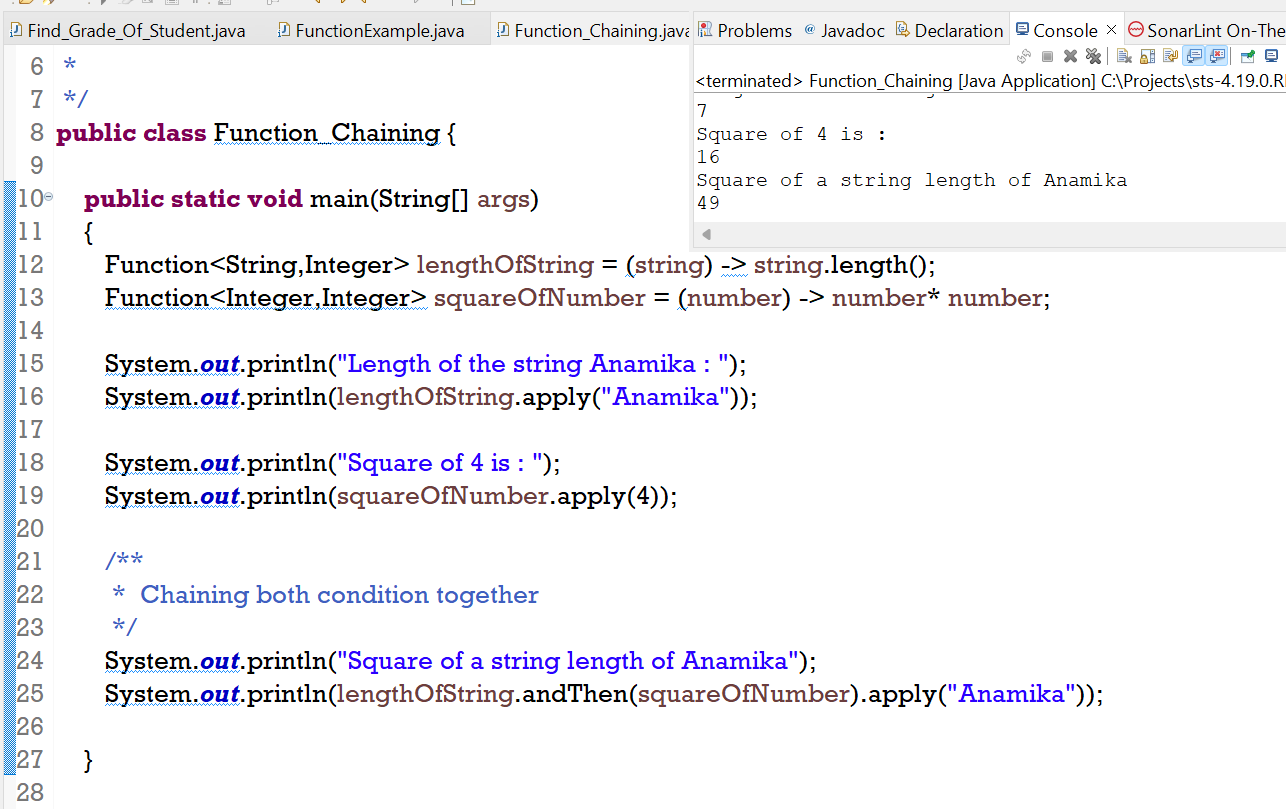
Let us suppose there are 2 functions f1 and f2 are there

* **andThen() : f1.andThen(f2) 🡪 f1 function operation will execute first and after that f2 function**

**will execute.**

* **compose() : f1.compose(f2) 🡪 f2 function operation will execute first and after that f1 function**

**will execute.**

****

**Consumer**

* Its an in-built functional interface in the **java.util.function** package.
* We use consumers when we need to consume objects, the consumer takes

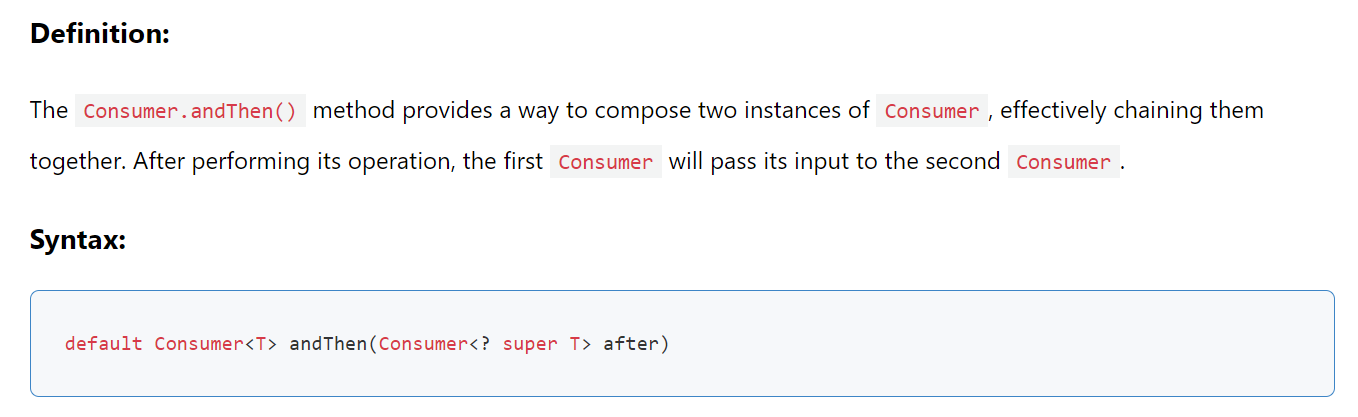
an input value and return nothing.

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Methods :**

* **void accept(T value)**
* **default Consumer<T> andThen(Consumer <? Super T> after)**

****

**A screenshot of a computer

AI-generated content may be incorrect.**

A screenshot of a computer

AI-generated content may be incorrect.

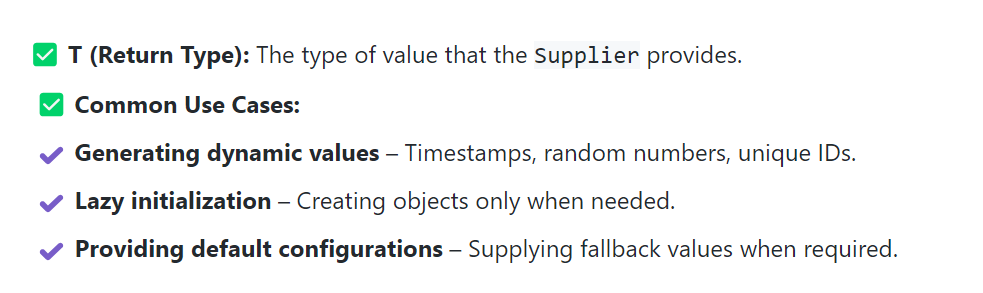
**Supplier**

* In Java **functional programming,** the Supplier<T> interface is a **functional interface that takes no**

**input but return a result** when called.

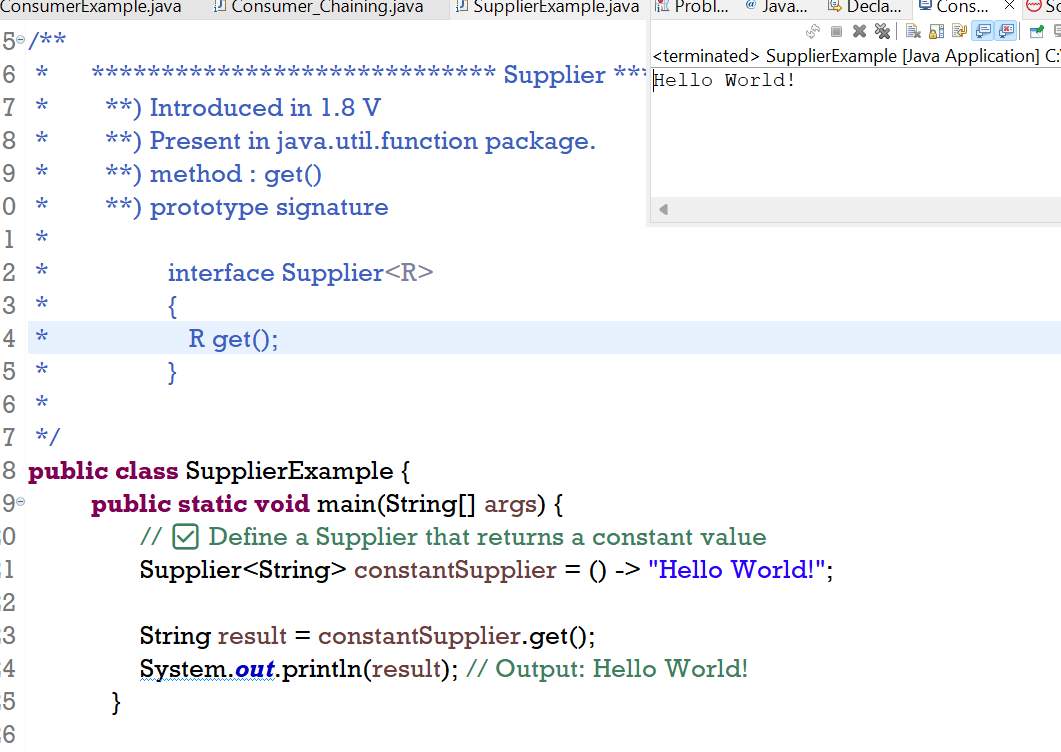
* Supplier interface **does not contain any default method or any static method** . So chaining is not

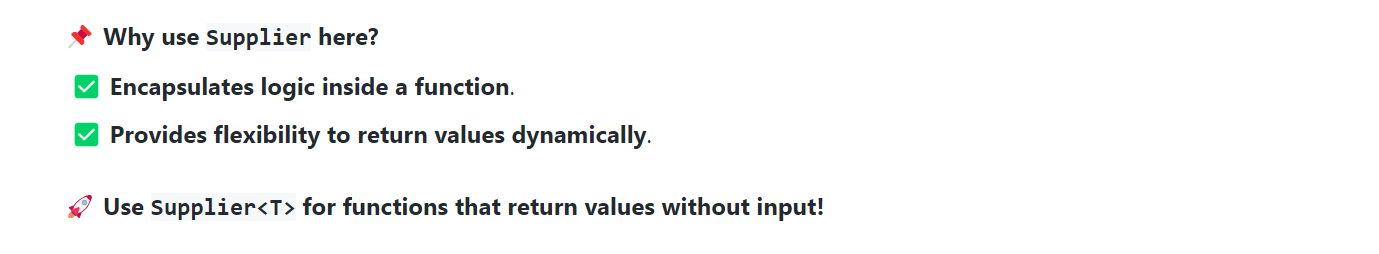
possible.



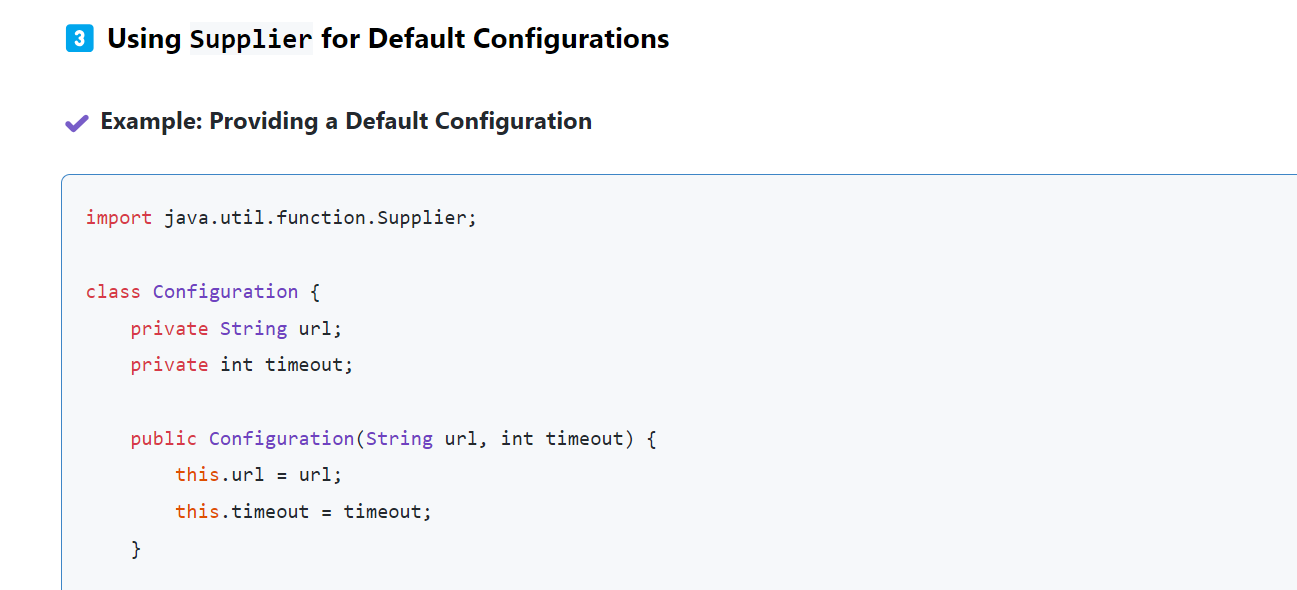
A screenshot of a computer code

AI-generated content may be incorrect.





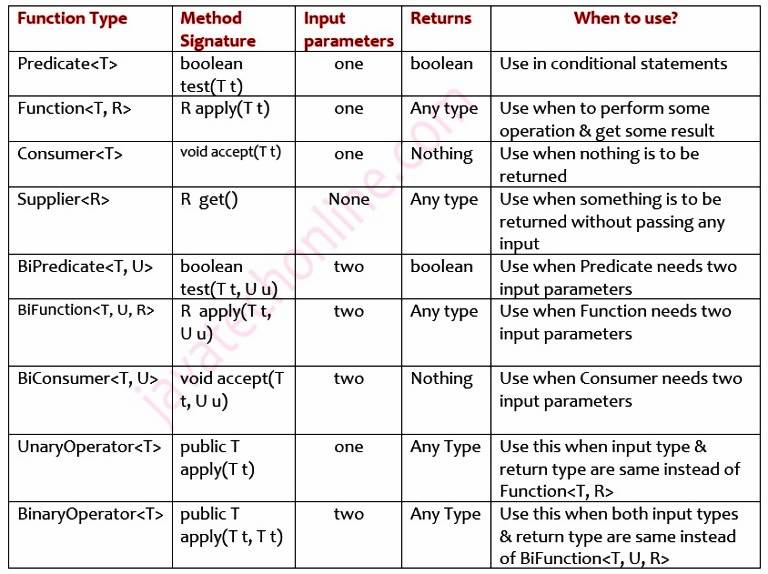




A computer code on a white background

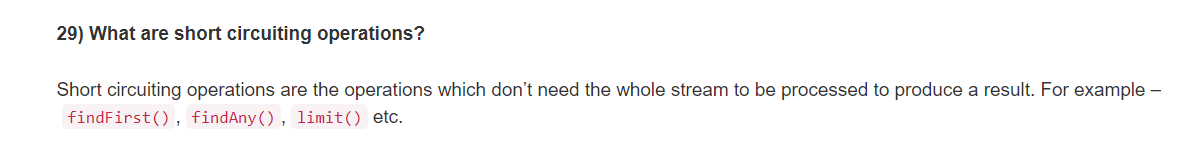
AI-generated content may be incorrect.

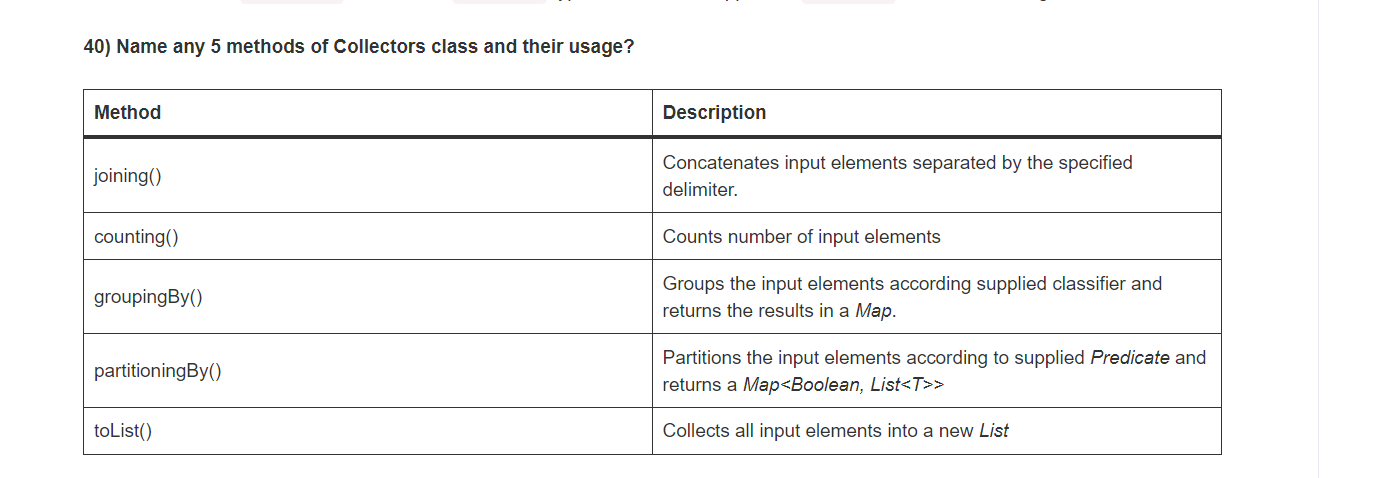




A screenshot of a computer

AI-generated content may be incorrect.





A screenshot of a computer program

AI-generated content may be incorrect.

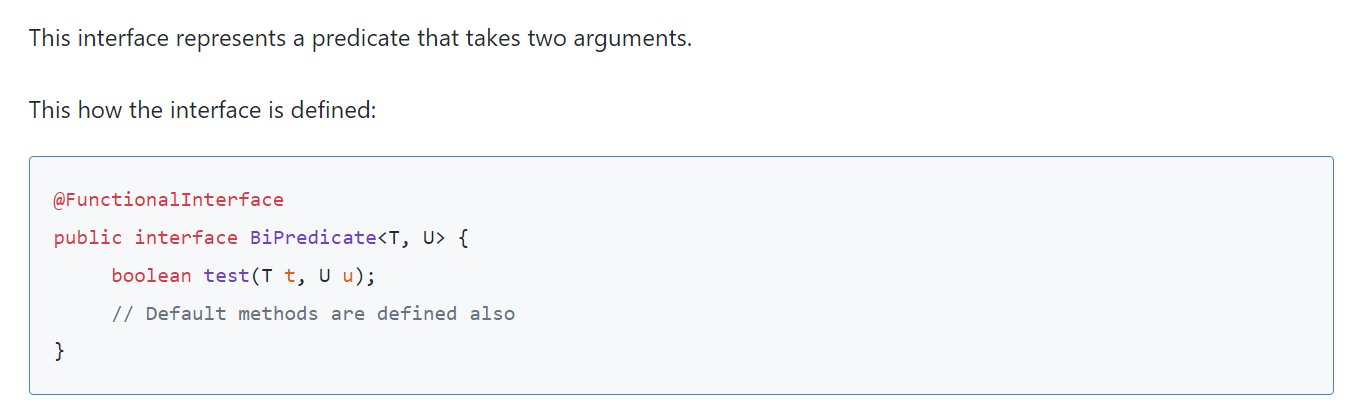
**Functional Interfaces having 2 input arguments**

**BiPredicate**

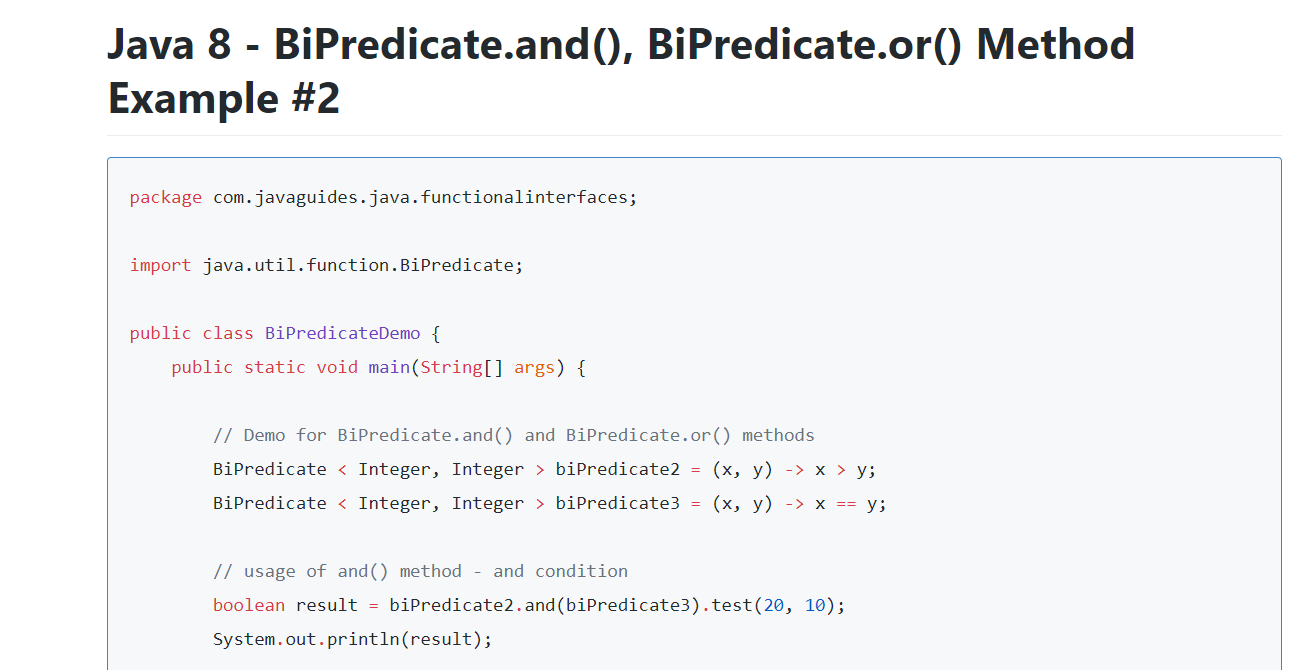
method : **test()** , **and()** ,**or(),** **negate()**

test : It is an abstract method

Other methods are default method.

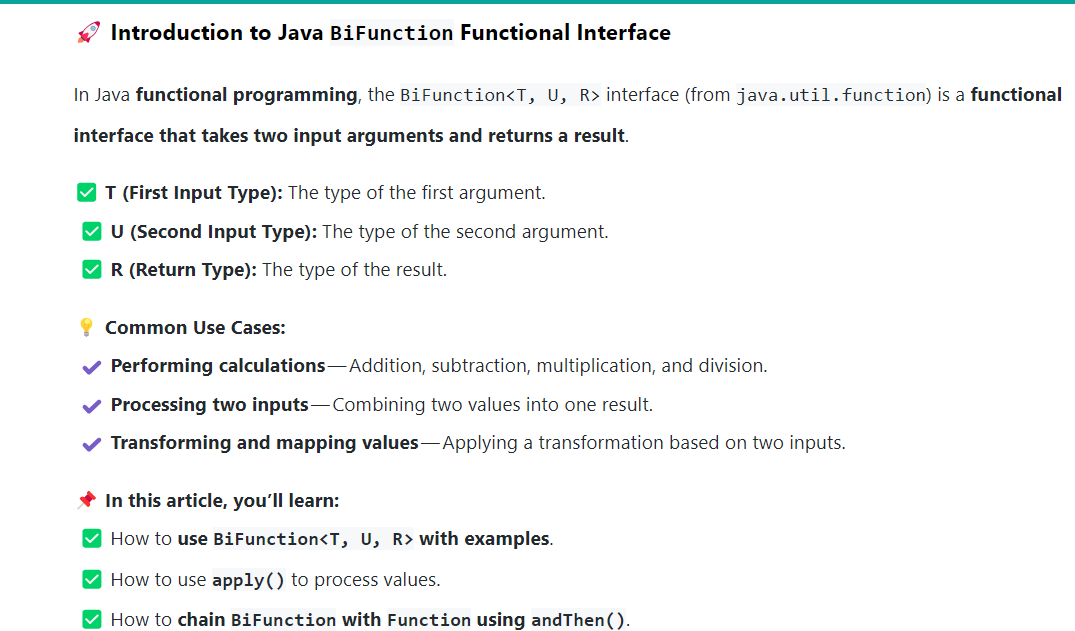
****

****

****

**BiFunction**

method : **apply()**, **andThen()**

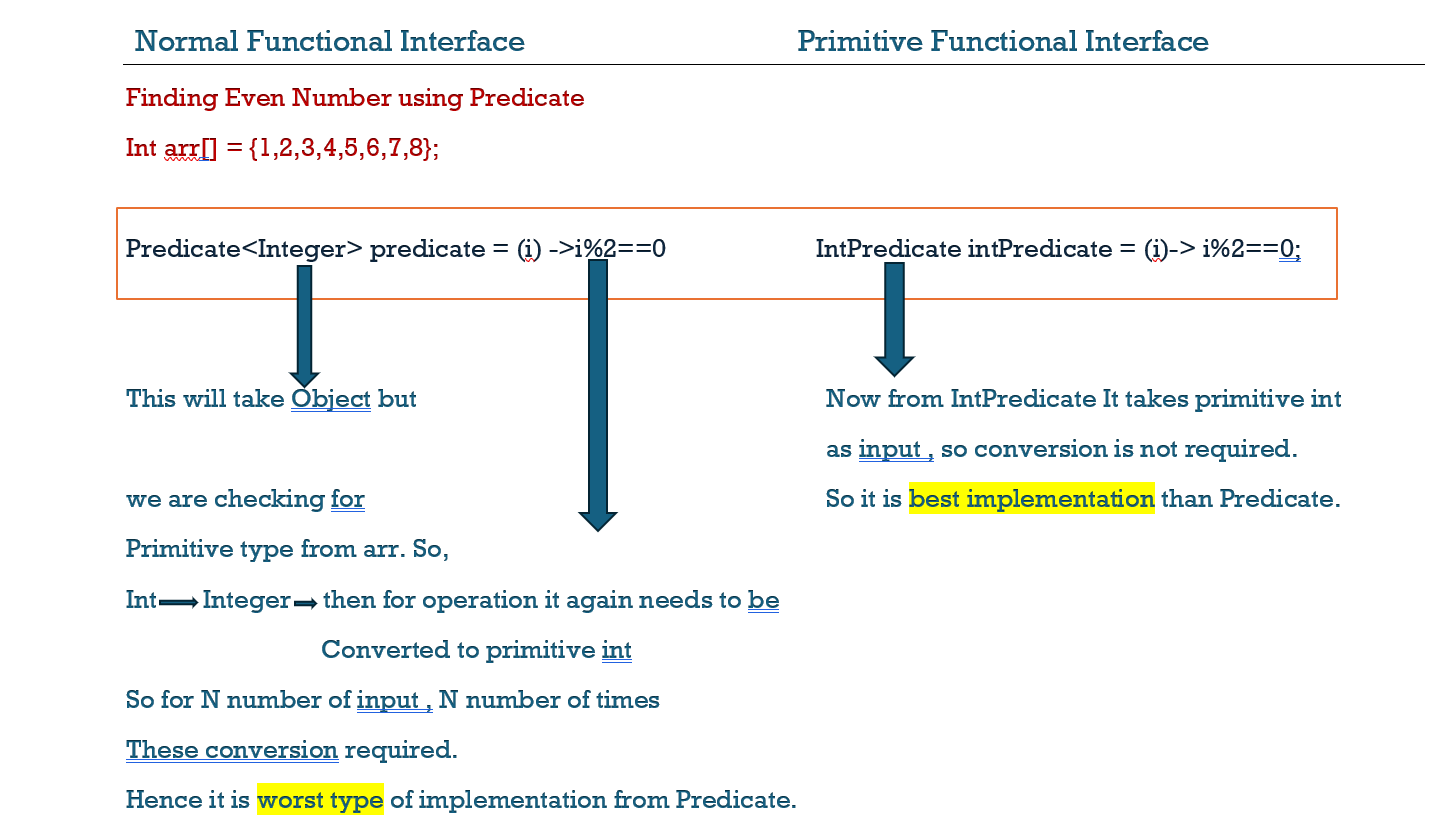
****

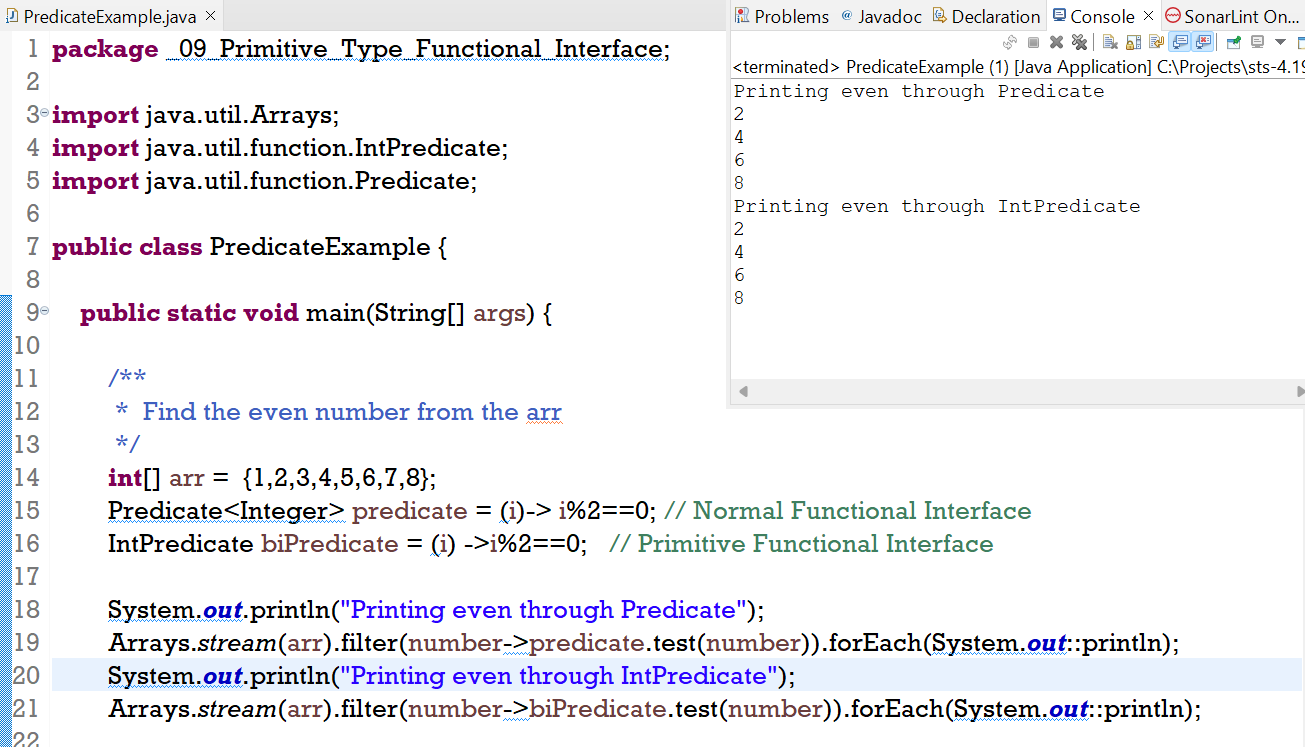
A screenshot of a computer

AI-generated content may be incorrect.

**Primitive Functional Interface for Predicate**

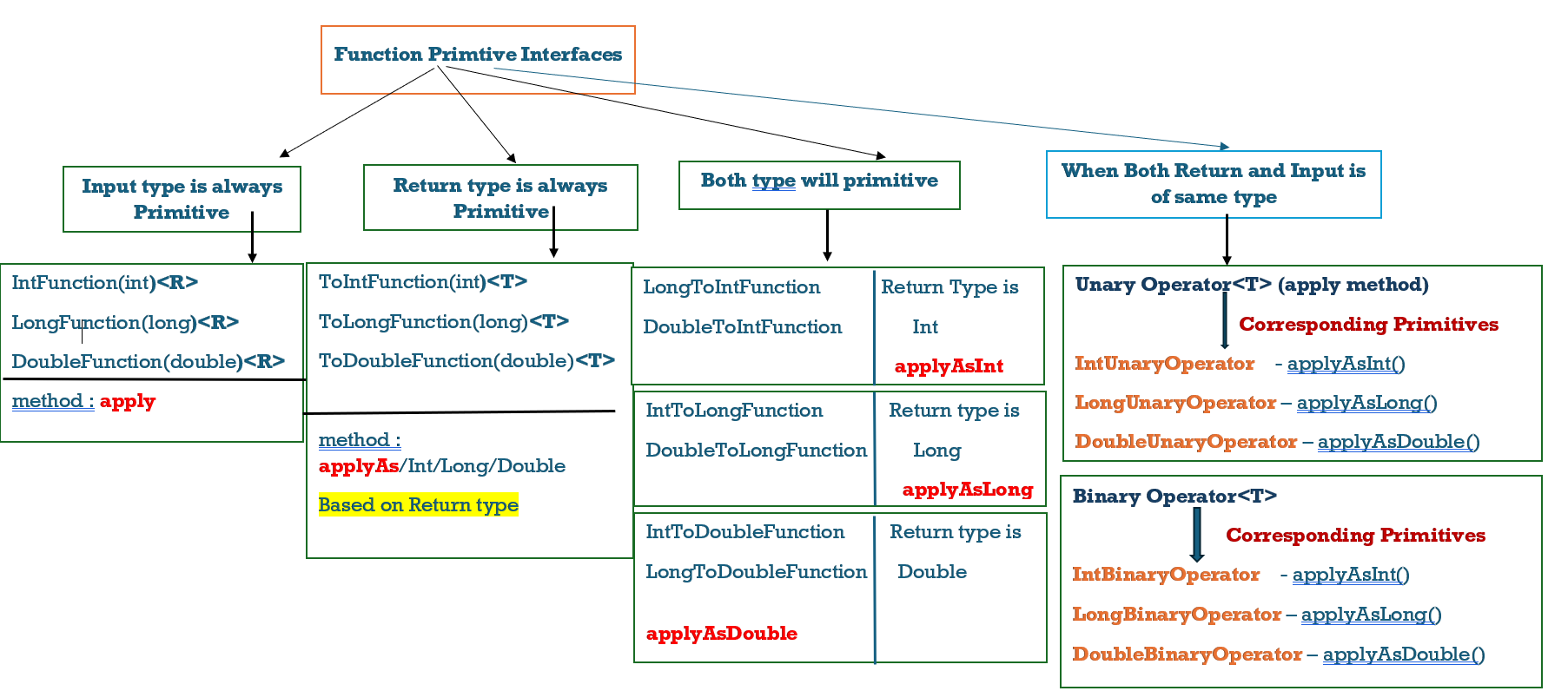
* **IntPredicate**
* **LongPredicate**
* **DoublePredicate**

****

****

**Primtive Functional Interfaces for Function**

**Different Primtive Type of Functional Functional Interface**



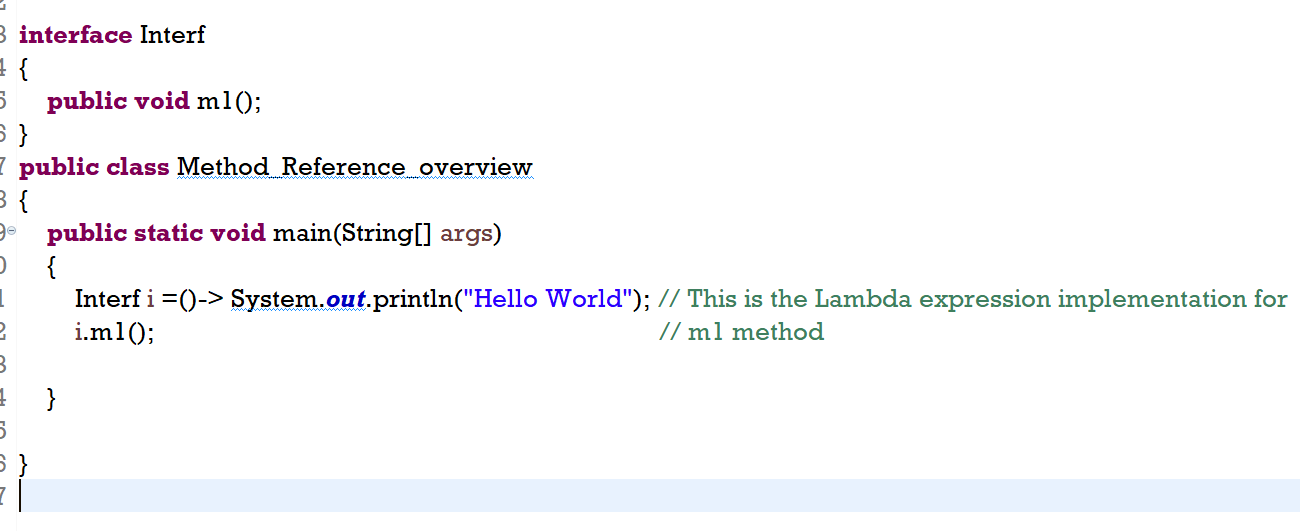
**UnaryOperator 🡪 One Input one Output and both of same type .**

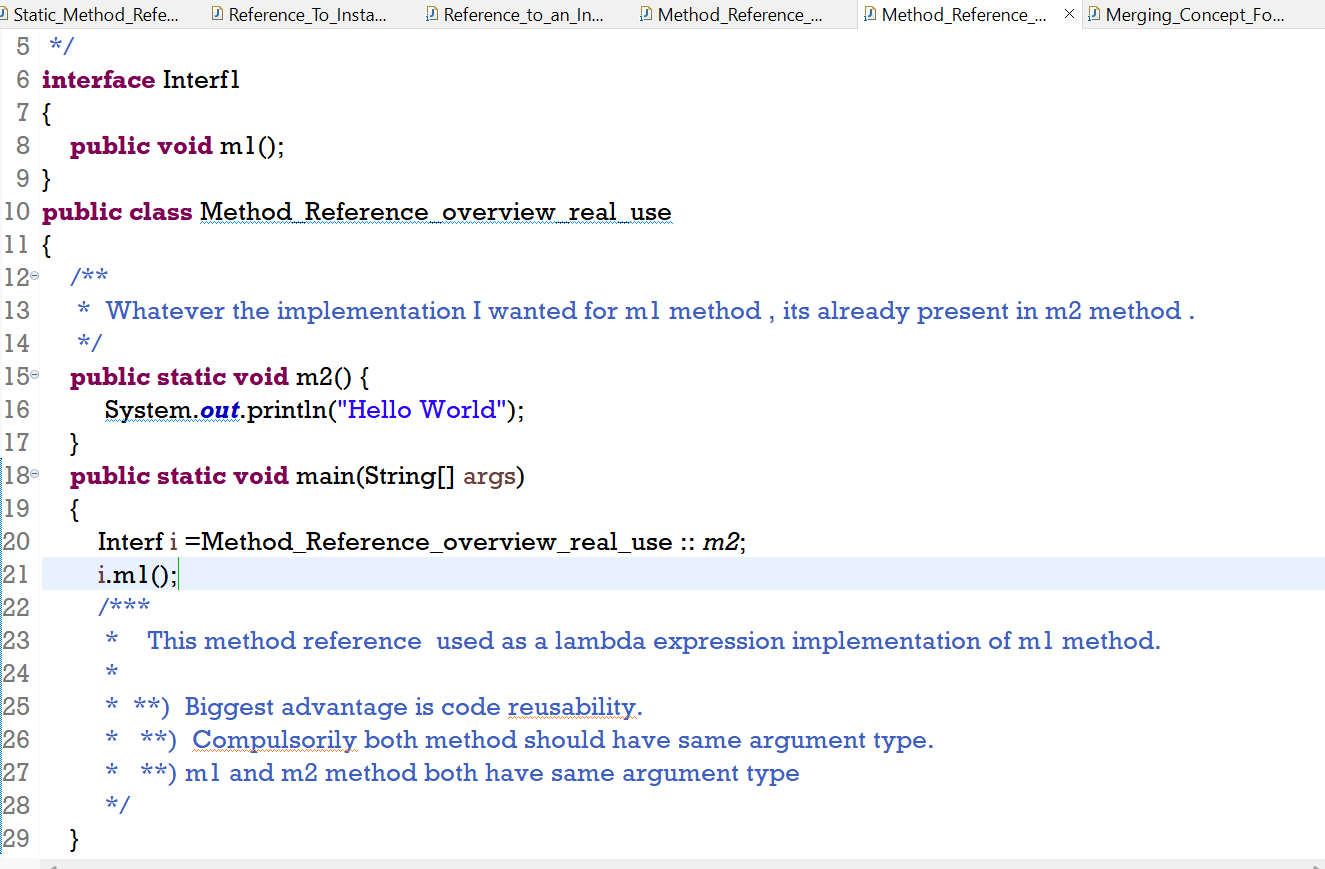
**BinaryOperator 🡪 Two Input and one Output all are of same type.**

**Method Reference**

* Method reference is used to refer method of the functional interface.
* It is a compact and easy form of lambda expression.
* Each time when you are using a lambda expression to just referring a method,

you can replace your lambda expression with a method reference.





**There are four kinds of method references:**

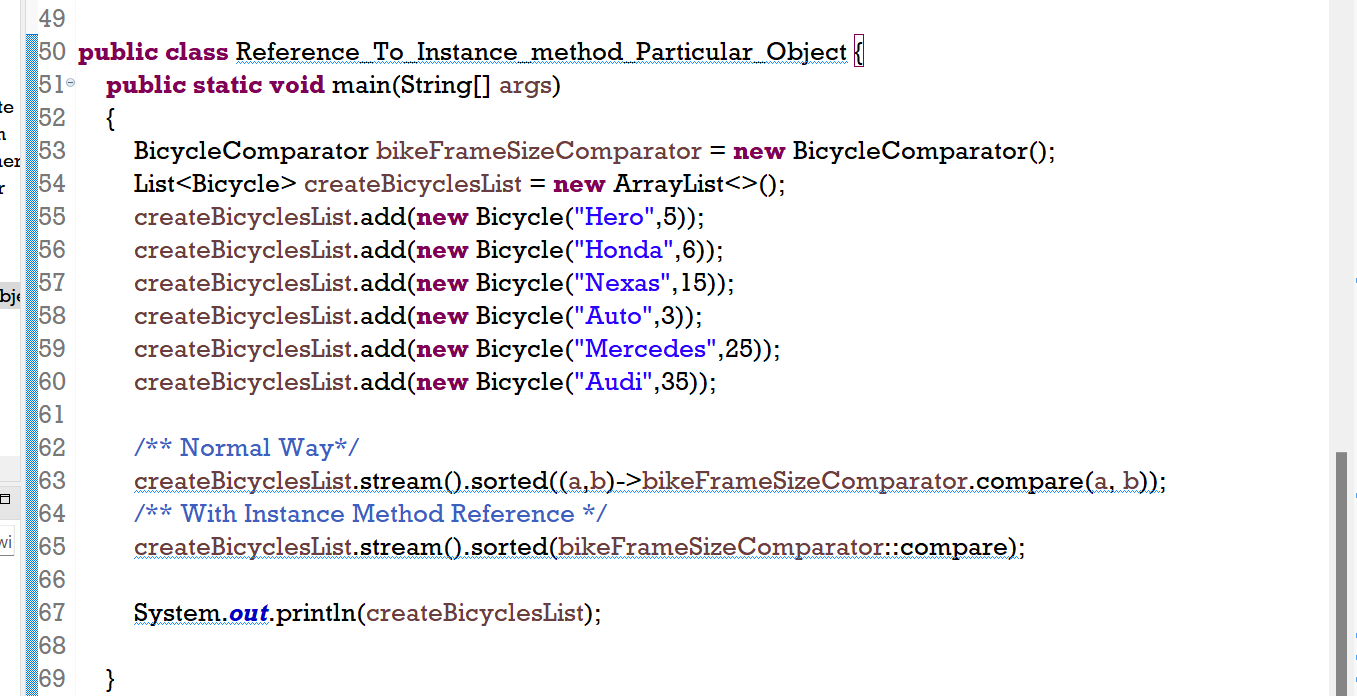
* Static Method Reference
* Instance Method Of Particular Object
* Instance Method Of Arbitrary Object of a Particular Type
* Constructor

**Static Method Reference**

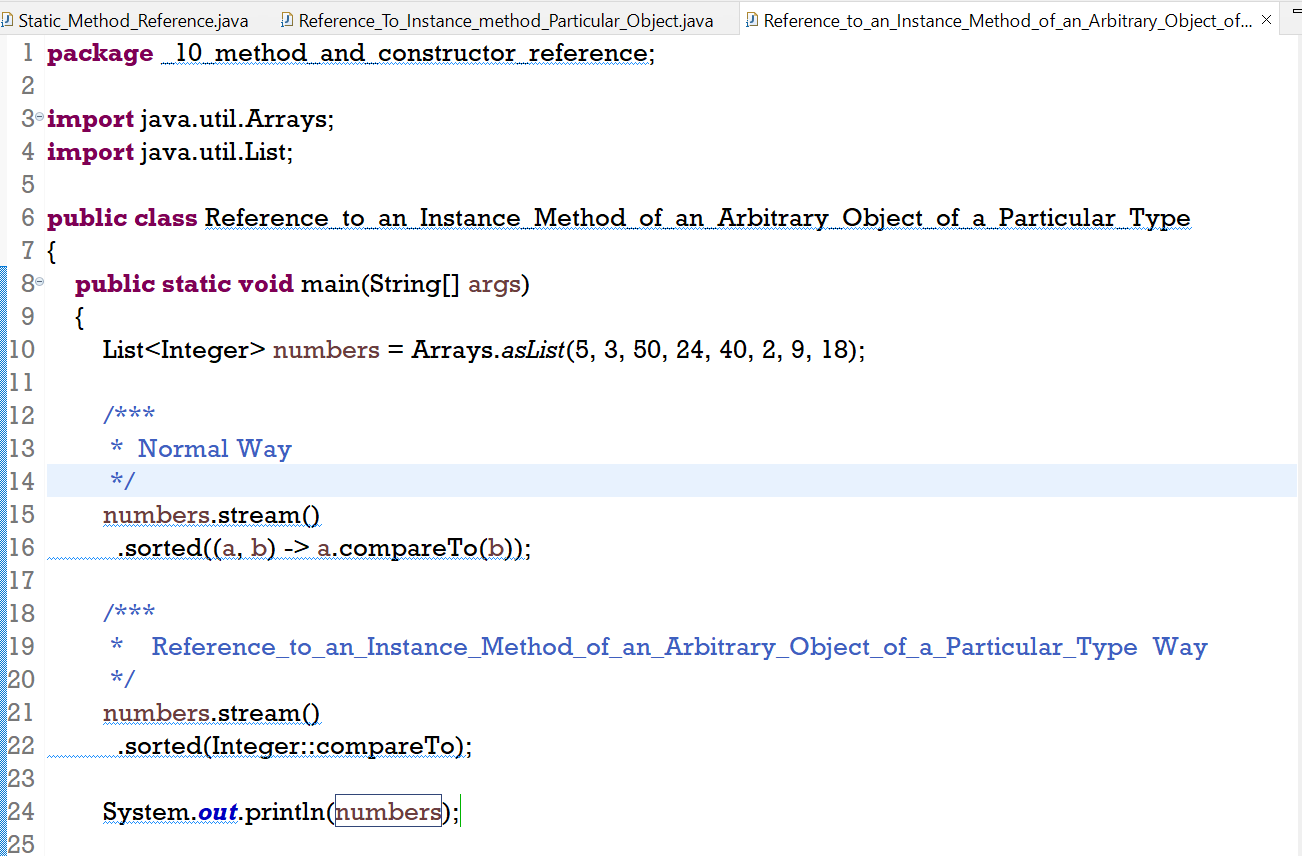
A screenshot of a computer program

AI-generated content may be incorrect.

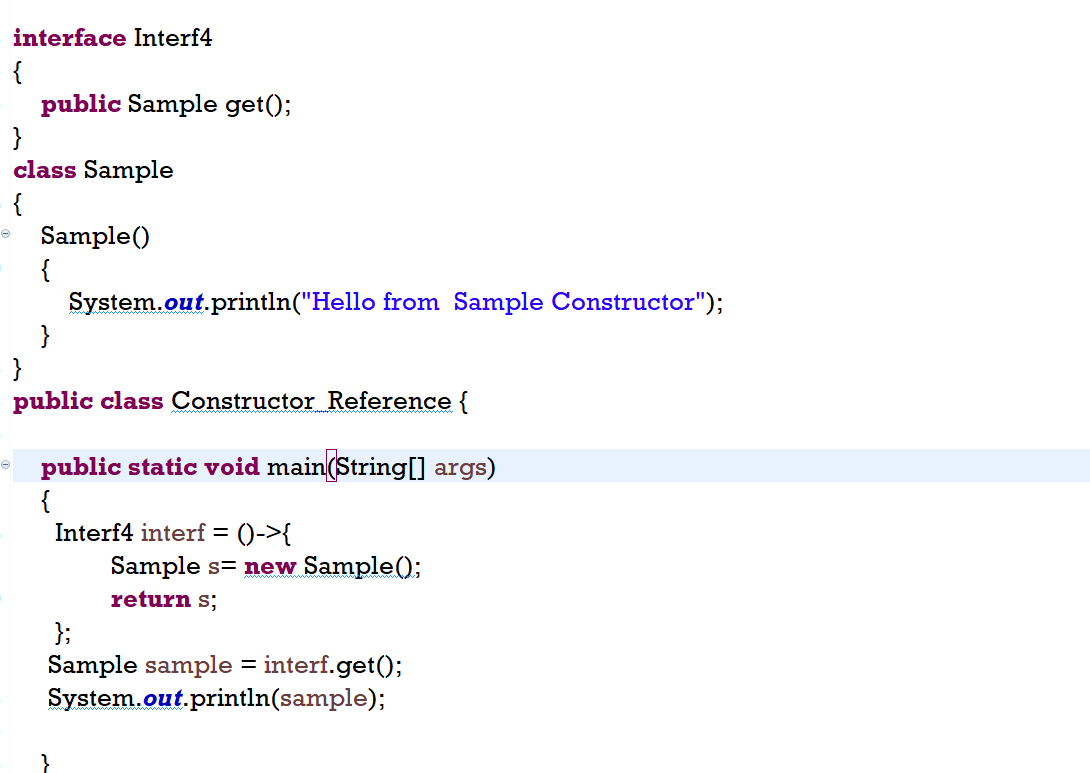
**Instance Method Reference**

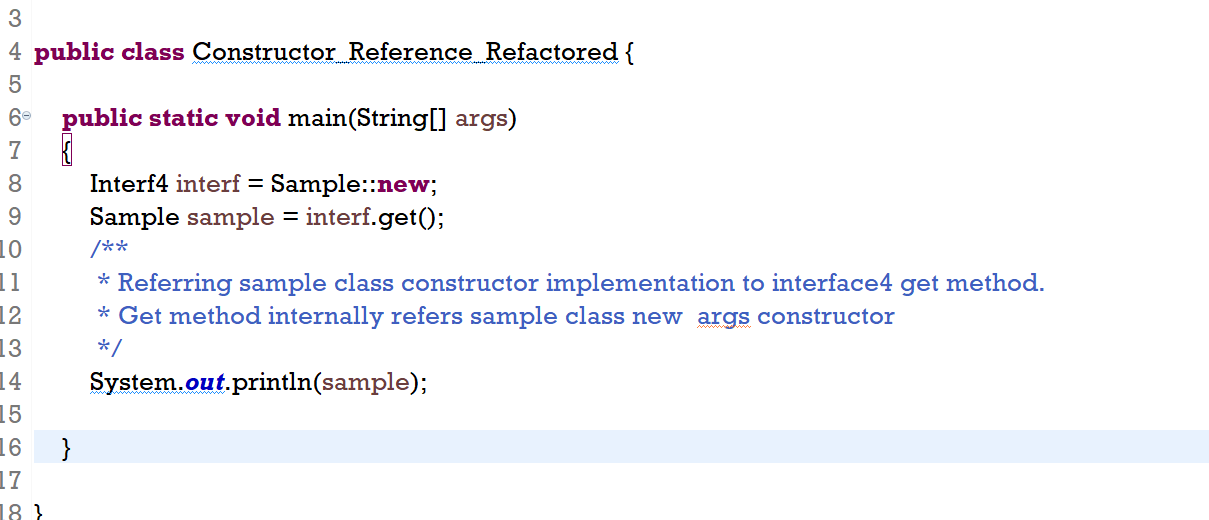


**Reference to an Instance Method of an Arbitrary Object of a Particular Type**



**Constructor Reference**

****

****

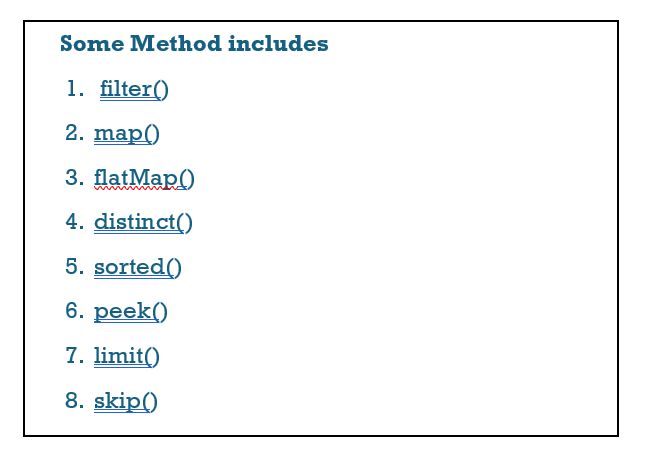
**Stream API**

* Stream operation is divided into **intermediate** and **terminal** operation.
* **Terminal** operation of Java Stream interface typically return **single** value.
* Terminal Operation cannot be chained together.

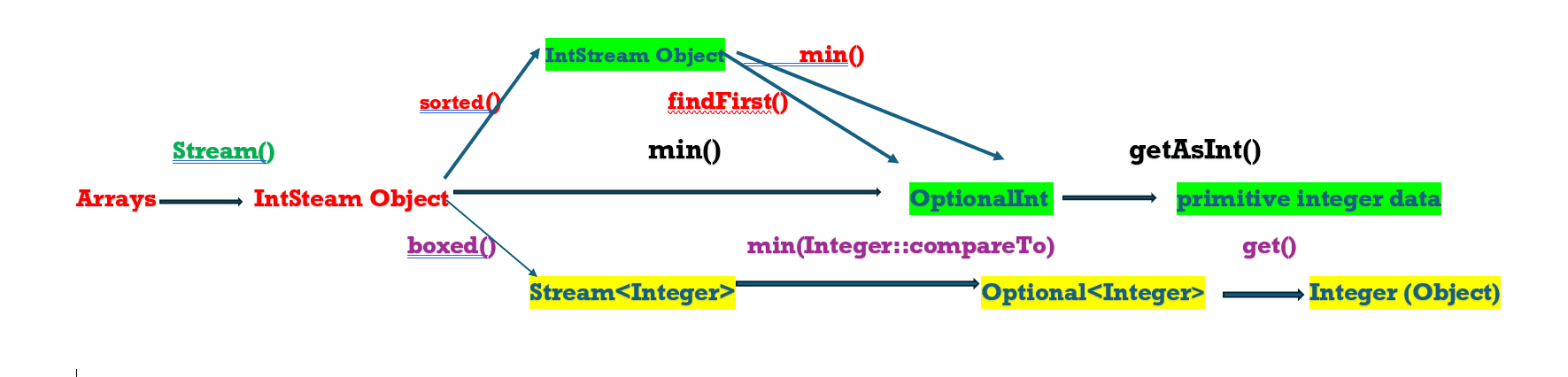
**A screen shot of a white sheet

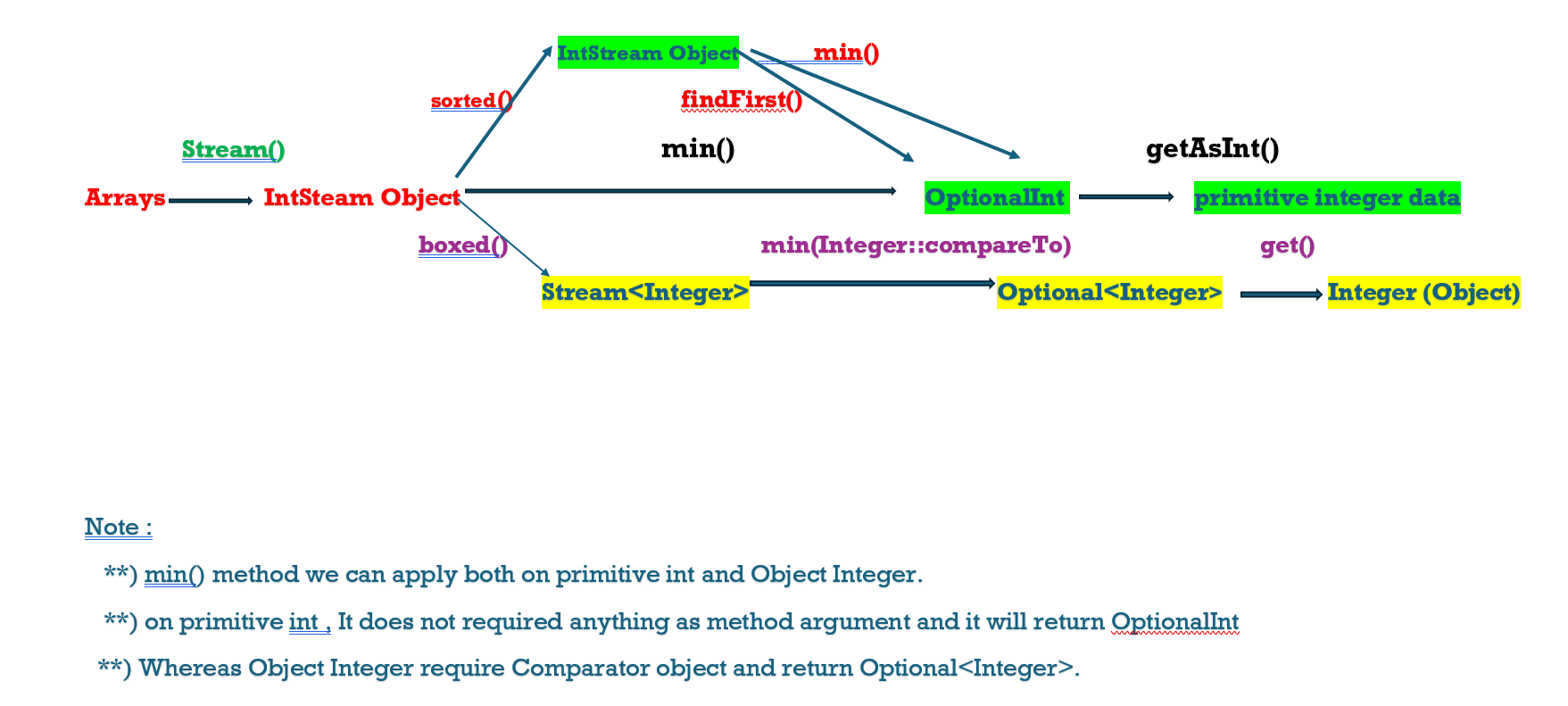
AI-generated content may be incorrect.**

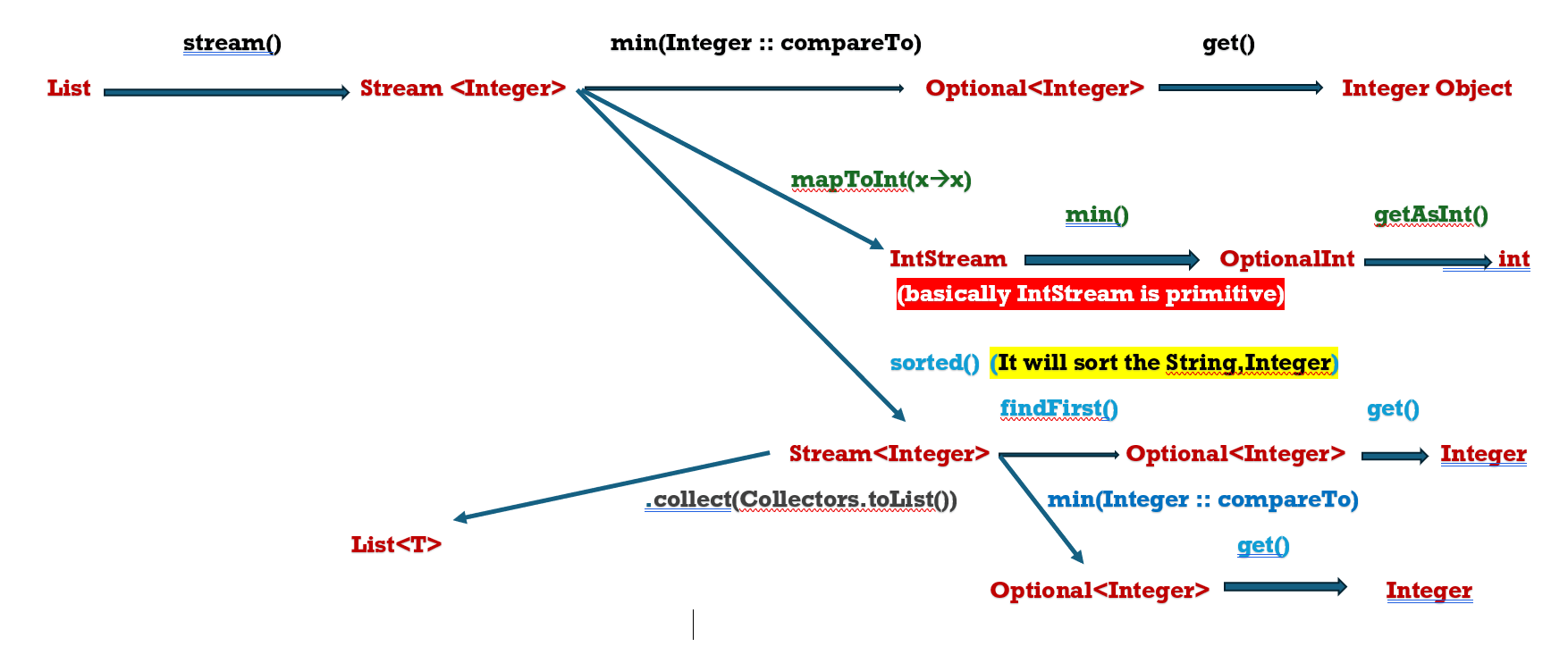
* Java 8 Streams support **intermediate operations** that **return another stream**.
* Allowing developers to chain multiple operation together to form the **pipeline**.
* Intermediate operations are lazy, meaning they are not executed until a terminal operation is invoked.



**Sorting**







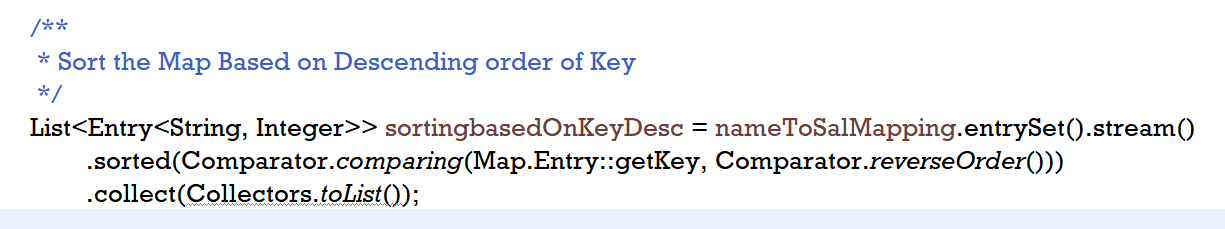
**Interview Question Link**

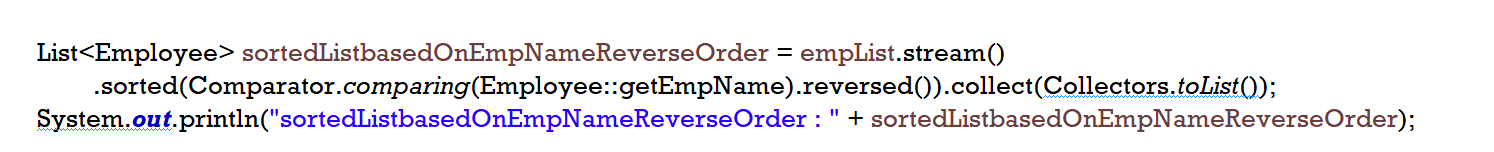
<https://www.javaguides.net/p/java-8-stream-api-tutorial.html>

**Employee related Java8 Questions :**

[**https://www.javaguides.net/2024/08/java-8-interview-questions-on-employee.html**](https://www.javaguides.net/2024/08/java-8-interview-questions-on-employee.html)

[**https://javaconceptoftheday.com/solving-real-time-queries-using-java-8-features-employee-management-system/#google\_vignette**](https://javaconceptoftheday.com/solving-real-time-queries-using-java-8-features-employee-management-system/#google_vignette)

****

****

**Note :** There are 2 images , One is operation on List and one is operation on map.

While Sorting reverse order from **Comparator.comparing**

**In case of map , we will have to provide Comparator.reverseOrder() inside Comparator.comparing method.**

**But**

**In case of List, we can provide .reversed() as a method call outside the Comprator.comparing**