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)

LambdaExpression.java

```
6    * Normal  way of writing method
7    */
8    public void displayMessage() {
9       System.out.println("Hello World");
10   }
11   /**
12    *    **) Lambda expression is an expression without name,
13    *          access-modifier,and return type
14    *    **) Lambda Expression of writing the same method
15    */
16
17      ()->{
18         System.out.println("Hello World");
19      }
20
```

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.

```
12  */
13
14 interface Interf
15 {
16     void m1();
17     /** Default method which is allowed in interface after 1.8 onwards  */
18     default void m2() {
19
20     }
21     /** Static method which is allowed in interface after 1.8 onwards  */
22     public static void m3() {
23
24     }
25
26     /** So this is an example of Functional Interface*/
27 }
28
29
```

Some of already existing Functional Interfaces

Runnable → run() method
Callable → call() method
Comparable → compareTo method

Functional_Interface.java

```
7  *     **) @FunctionalInterface is also an annota
8  *     **) It is also called SAM (Single Abstract Me
9  *
10 *
11 *
12 */
13
14 @FunctionalInterface
15 interface Interf
16 {
17     public void m1();
18     public void m2();
19 }
20
21
```

Problems × | Javadoc | Declaration | Console | SonarLint On-The-Fly

1 error, 0 warnings, 6 others

Description

- Errors (1 item)
  - Invalid '@FunctionalInterface' annotation; Interf is not a functional interface
- Infos (6 items)

Functional_Interface.java

```
 7  *     **) @FunctionalInterface is also an annota
 8  *     **) It is also called SAM (Single Abstract Me
 9  *
10  *
11  *
12  */
13
14 @FunctionalInterface
15 interface Interf
16 {
17
18 }
19
20
21
22
```

Problems | Javadoc | Declaration | Console | SonarLint On-The-Fly

1 error, 0 warnings, 6 others

Description

- Errors (1 item)
  - Invalid '@FunctionalInterface' annotation; Interf is not a functional interface
- Infos (6 items)

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

Functional Interface W.R.T Inheritance

```
13
14 @FunctionalInterface
15 interface Interf
16 {
17   public void m1();
18 }
19
20 @FunctionalInterface
21 interface Inter1 extends Interf
22 {
23
24 }
25
```

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

```
 *     **) @FunctionalInterface is also an annota
 *     **) It is also called SAM (Single Abstract Me
 *
 *
 *
 */

@FunctionalInterface
interface Interf
{
  public void m1();
}

@FunctionalInterface
interface Inter1 extends Interf
{
   public void m1();
}
```

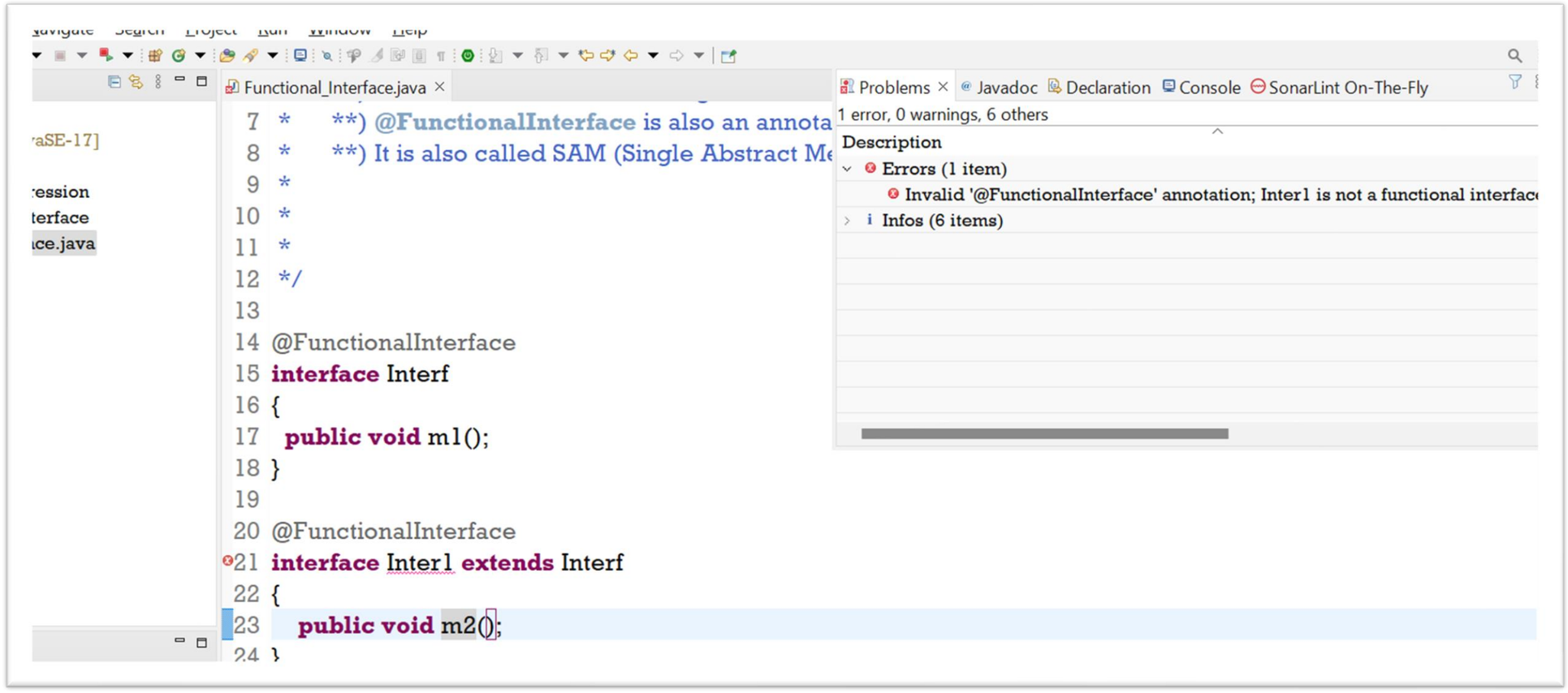
6 items

Description

Infos (6 items)

to match (incl.

➔ 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

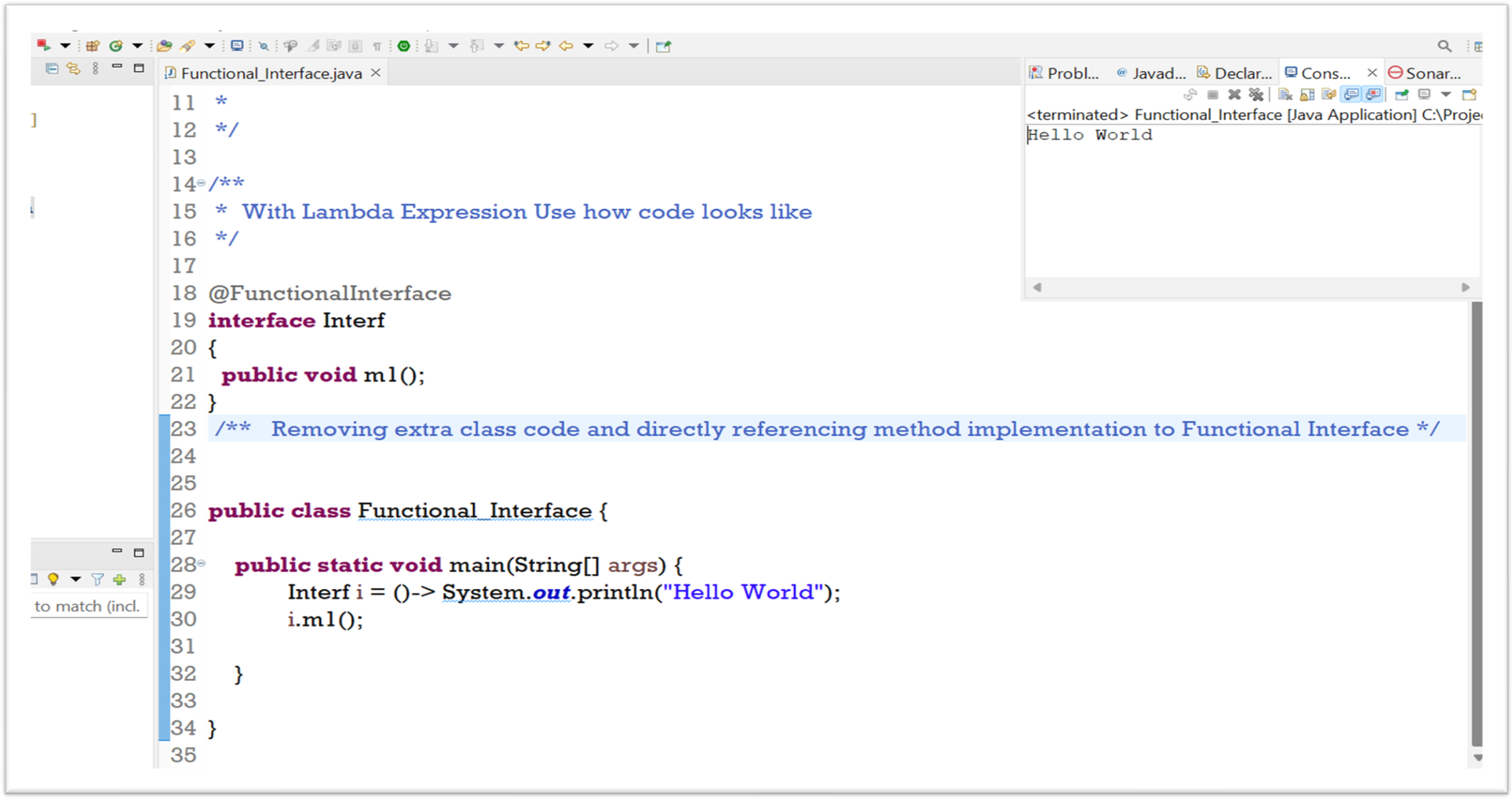
Functional_Interface.java

```
/**
 *  Without Lambda Expression Use how code looks like
 */

@FunctionalInterface
interface Interf
{
  public void m1();
}
 /**    This is an extra class we are using just to provide implementation to interface method */

class Demo implements Interf
{
   public void m1() {
      System.out.println("Hello World");
   }
}

public class Functional_Interface {

   public static void main(String[] args) {
         Interf i = new Demo(); //Runtime polymorphism
         i.m1();

   }
```

Probl... Javad... Declar... Cons... Sonar...

<terminated> Functional_Interface [Java Application] C:\Pro

```
Hello World
```


Functional_Interface.java
Probl...
Javad...
Declar...
Cons...
Sonar...
<terminated> Functional_Interface [Java Application] C:\Proje
Hello World
to match (incl.
11 *
12 */
13
14 /**
15 * With Lambda Expression Use how code looks like
16 */
17
18 @FunctionalInterface
19 interface Interf
20 {
21 public void m1();
22 }
23 /** Removing extra class code and directly referencing method implementation to Functional Interface */
24
25
26 public class Functional_Interface {
27
28 public static void main(String[] args) {
29 Interf i = ()-> System.out.println("Hello World");
30 i.m1();
31
32 }
33
34 }
35

Functional_Interface.java

```
15  * Creating Thread using Runnable without lambda Expression
16  */
17 class MyRunnable implements Runnable
18 {
19     @Override
20     public void run() {
21         for (int i = 0; i < 10; i++) {
22             System.out.println("Child Thread");
23         }
24     }
25 }
26
27 public class Functional_Interface {
28
29     public static void main(String[] args) {
30         Runnable r = new MyRunnable();
31         Thread t = new Thread(r);
32         t.start();
33         for(int i=0;i<10;i++) {
34             System.out.println("Parent Thread");
35         }
36     }
```

Probl... Javad... Declar... Cons... Sonar...

<terminated> Functional_Interface [Java Application] C:\Prc

```
Parent Thread
Parent Thread
Parent Thread
Parent Thread
Parent Thread
Child Thread
Child Thread
Child Thread
```

names to match (incl.

Spring Java Reconcile

```
 9  *
10  *
11  *
12  */
13
14 /**
15  *  Creating Thread using Runnable with lambda Expression
16  */
17 public class Functional_Interface {
18
19     public static void main(String[] args) {
20         Runnable r = () -> {
21             for (int i = 0; i < 10; i++) {
22                 System.out.println("Child Thread");
23             }
24         };
25         Thread t = new Thread(r);
26         t.start();
27         for (int i = 0; i < 10; i++) {
28             System.out.println("Parent Thread");
29         }
30     }
```

```
<terminated> Functional_Interface [Java Application] C:\Pro
Parent Thread
Parent Thread
Parent Thread
Parent Thread
Parent Thread
Parent Thread
Child Thread
Child Thread
```

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.

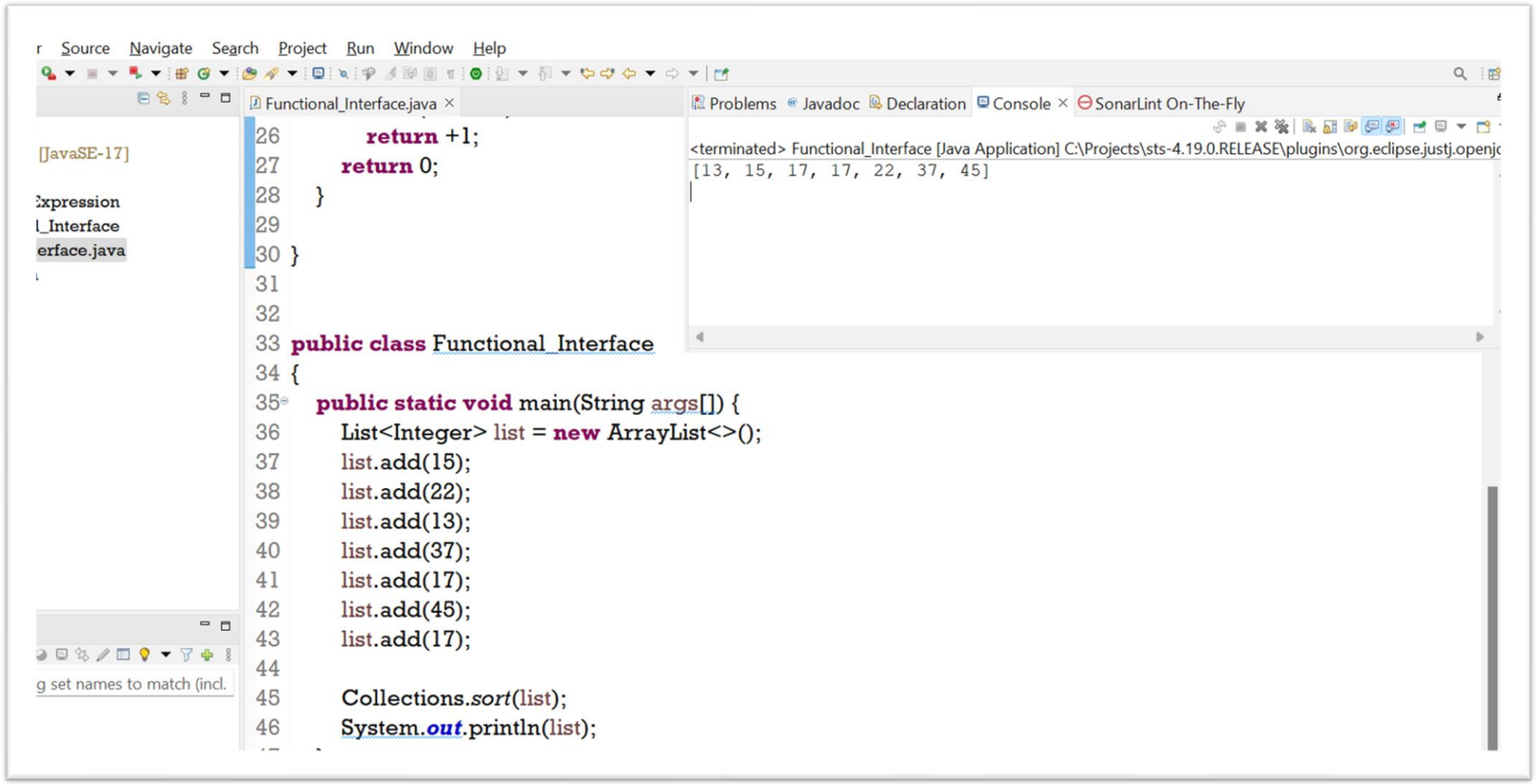
Interface → Comparator

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

Functional_Interface.java

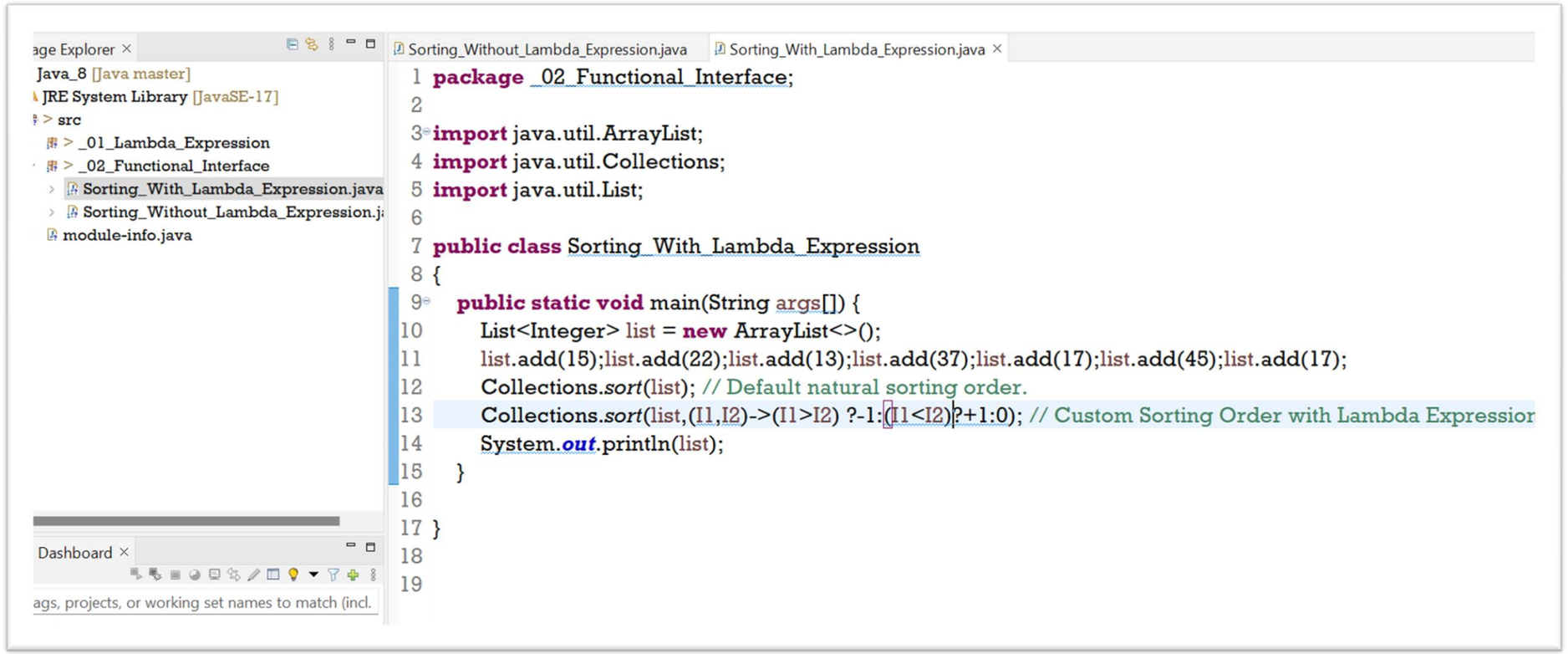
```
17 */
18 /**
19  *  Custom Sorting Order (Descending Order)
20  */
21 class MyComparator implements Comparator<Integer>{
22
23    @Override
24    public int compare(Integer o1, Integer o2) {
25       if (o1 < o2)
26          return +1;
27       else if (o1 > o2)
28          return -1;
29       return 0;
30    } }
31 public class Functional_Interface
32 {
33    public static void main(String args[]) {
34       List<Integer> list = new ArrayList<>();
35       list.add(15);list.add(22);list.add(13);list.add(37);list.add(17);list.add(45);list.add(17);
36
37       Collections.sort(list, new MyComparator());
38       System.out.println(list);
39    }
40
```

Problems | Javadoc | Declaration | Console | SonarLint On-...

<terminated> Functional_Interface [Java Application] C:\Projects\sts-4.19.0.REL

```
[45, 37, 22, 17, 17, 15, 13]
```

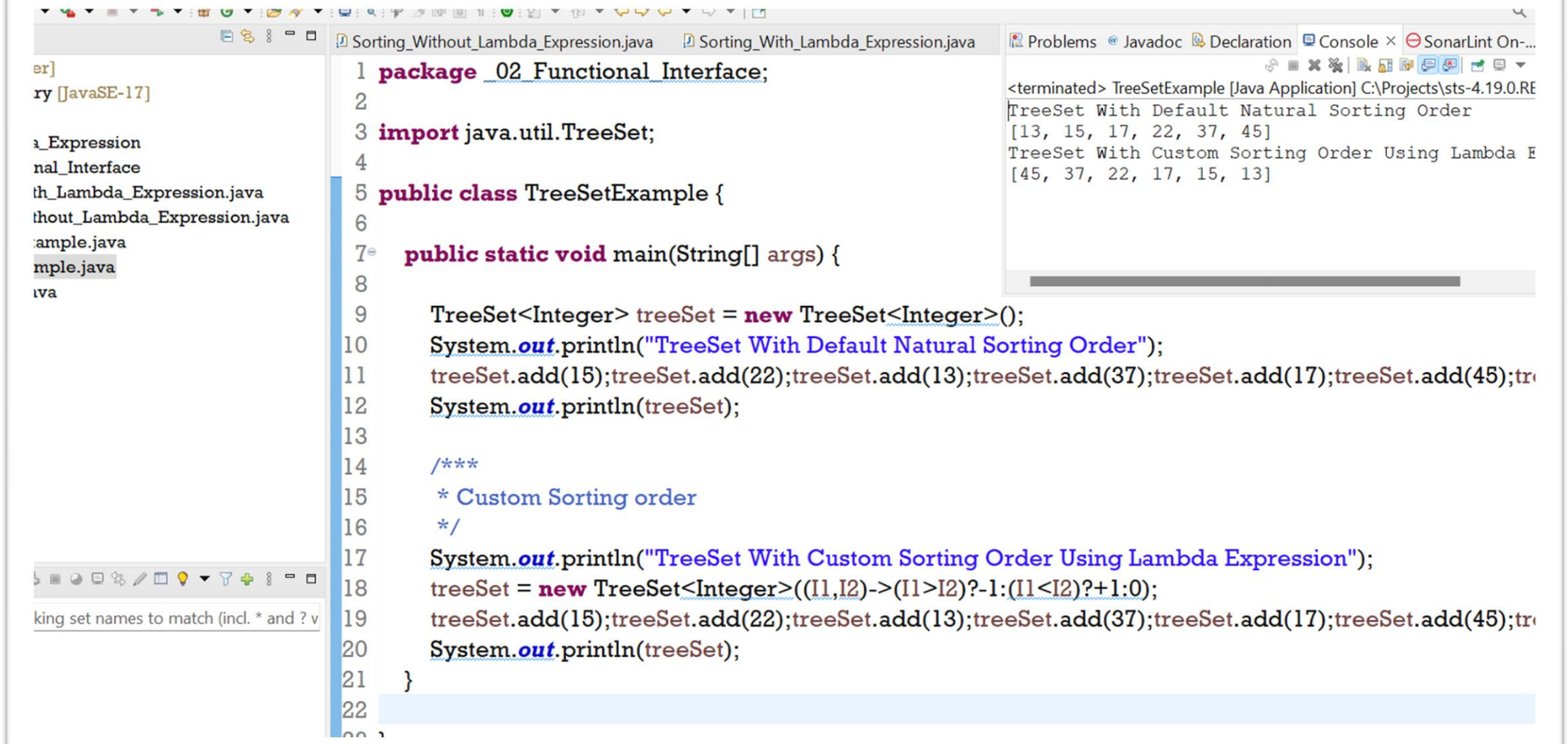
Sorting of Array List with using Lambda Expression.



Sorting In Tree Set

Sorting_Without_Lambda_Expression.java | Sorting_With_Lambda_Expression.java | *TreeSetExample.java

```java
package _02_Functional_Interface;

import java.util.TreeSet;

public class TreeSetExample {

    public static void main(String[] args) {

        TreeSet<Integer> treeSet = new TreeSet<Integer>(); // This is meant for Default Natural Sorting order
        TreeSet<Integer> treeSet2 = new TreeSet<Integer>(new Comparator()); // This is meant for Custom Sorting order

    }

}
```

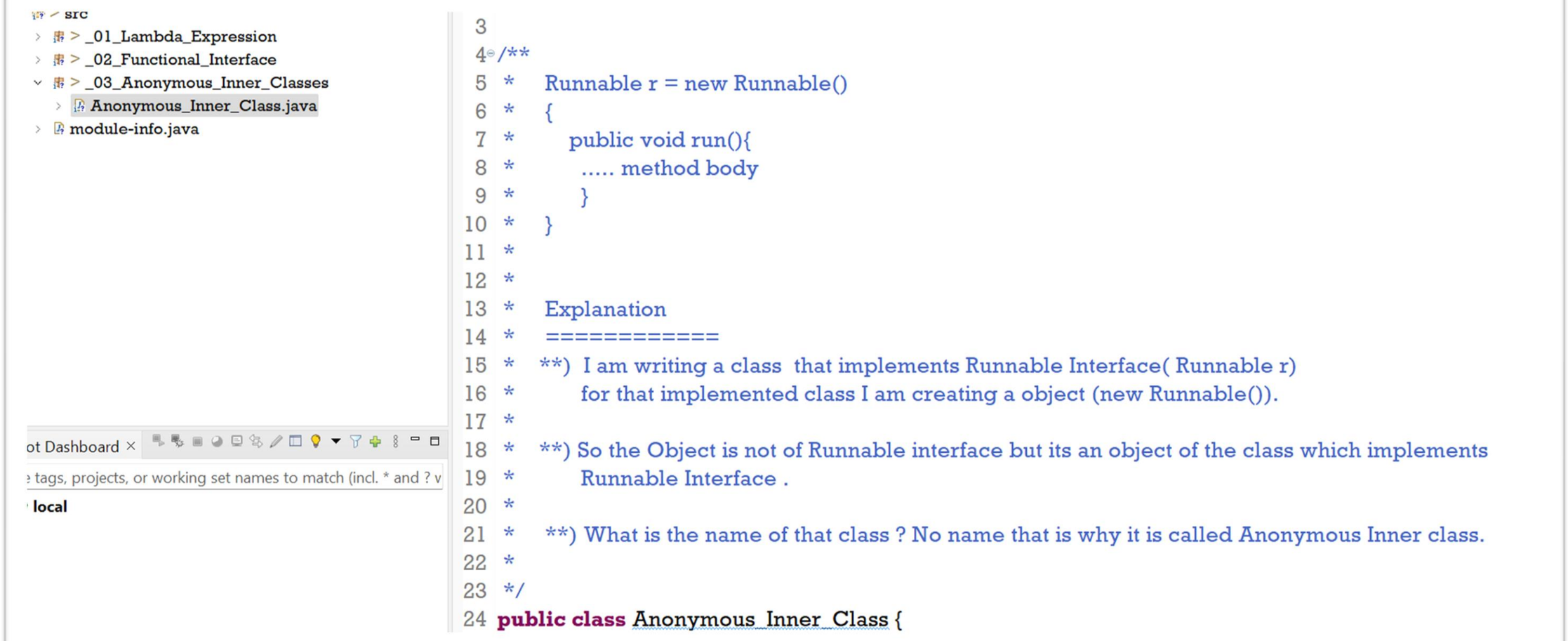

er]
ry [JavaSE-17]
a_Expression
nal_Interface
th_Lambda_Expression.java
thout_Lambda_Expression.java
ample.java
mple.java
va
king set names to match (incl. * and ? v
Sorting_Without_Lambda_Expression.java
Sorting_With_Lambda_Expression.java
1 package _02_Functional_Interface;
2
3 import java.util.TreeSet;
4
5 public class TreeSetExample {
6
7 public static void main(String[] args) {
8
9 TreeSet<Integer> treeSet = new TreeSet<Integer>();
10 System.out.println("TreeSet With Default Natural Sorting Order");
11 treeSet.add(15);treeSet.add(22);treeSet.add(13);treeSet.add(37);treeSet.add(17);treeSet.add(45);tr
12 System.out.println(treeSet);
13
14 /***
15 * Custom Sorting order
16 */
17 System.out.println("TreeSet With Custom Sorting Order Using Lambda Expression");
18 treeSet = new TreeSet<Integer>((I1,I2)->(I1>I2)?-1:(I1<I2)?+1:0);
19 treeSet.add(15);treeSet.add(22);treeSet.add(13);treeSet.add(37);treeSet.add(17);treeSet.add(45);tr
20 System.out.println(treeSet);
21 }
22
Problems
Javadoc
Declaration
Console
SonarLint On-...
<terminated> TreeSetExample [Java Application] C:\Projects\sts-4.19.0.RE
TreeSet With Default Natural Sorting Order
[13, 15, 17, 22, 37, 45]
TreeSet With Custom Sorting Order Using Lambda E
[45, 37, 22, 17, 15, 13]

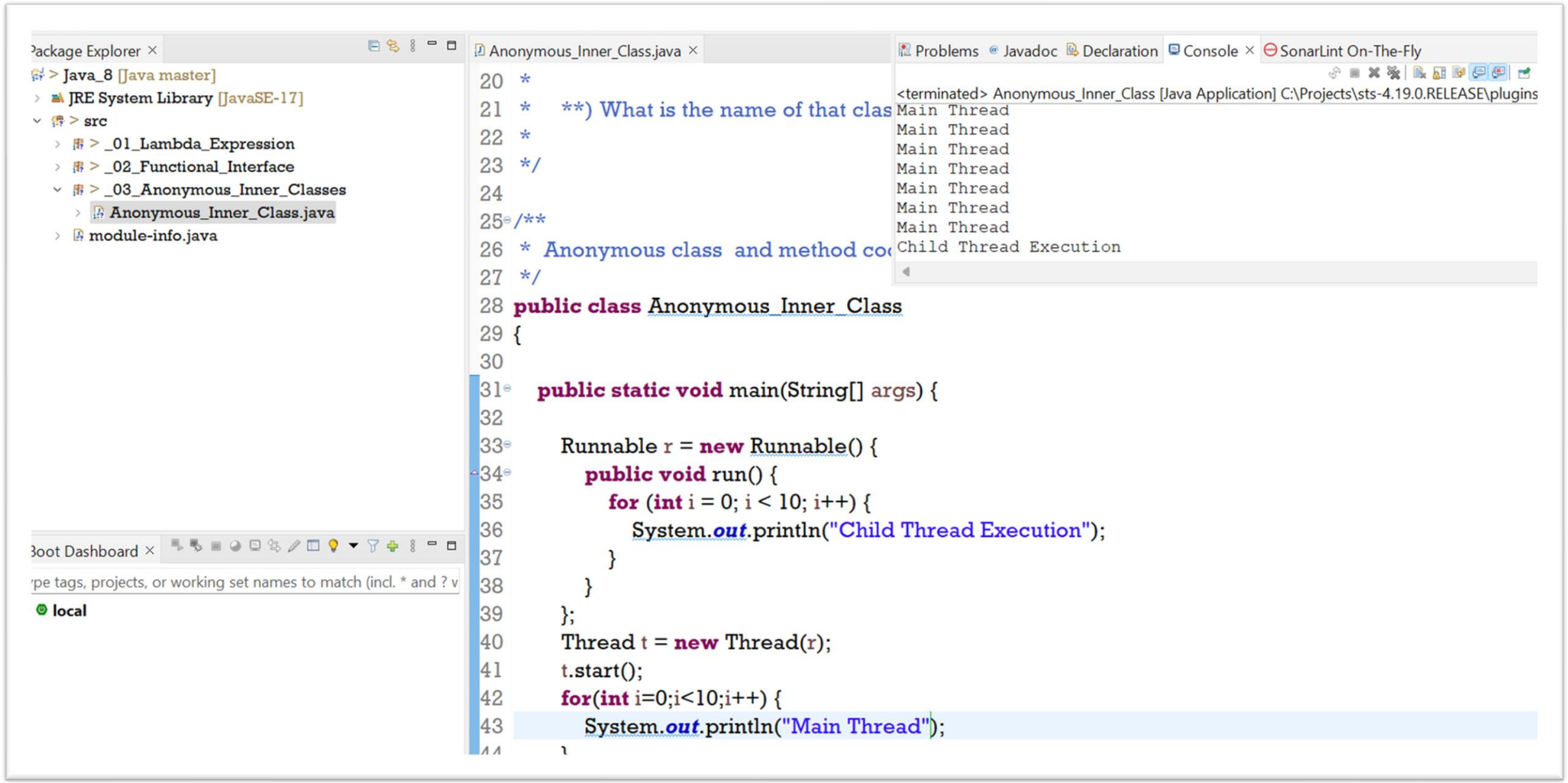
Sorting on Custom Object (Employee object)

```
 35    }
 36
 37 }
 38 public class Employee_Sorting {
 39    public static void main(String[] args)
 40    {
 41     ArrayList<Employee> listEmployee = new ArrayLis
 42     listEmployee.add(new Employee(510,"Karan"));
 43     listEmployee.add(new Employee(435,"Karan"));
 44     listEmployee.add(new Employee(720,"Karan"));
 45     listEmployee.add(new Employee(414,"Karan"));
 46     listEmployee.add(new Employee(329,"Karan"));
 47     listEmployee.add(new Employee(580,"Karan"));
 48
 49     System.out.println(listEmployee);
 50
 51     /**  Sort Based on EmpId  using Lambda Expression */
 52     Collections.sort( listEmployee,(e1,e2)->
 53                     (e1.getEmpno() > e2.getEmpno()) ? +1
 54                     : (e1.getEmpno() <e2.getEmpno()) ?-1 : 0
 55                     );
 56
 57     System.out.println("After Sorting based on empId ascending order : ");
 58     System.out.println(listEmployee);
```

```
<terminated> Employee_Sorting [Java Application] C:\Projects\sts-4.19.0.RELEA
[Employee [empno=510, ename=Karan], Employee [empl
After Sorting based on empId ascending order :
[Employee [empno=329, ename=Karan], Employee [empl
```

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**

Anonymous Inner Class	Lambda Expression
It is class without name	It is method without name. (Anonymous function)
It can extend abstract and concrete class	It can not extend abstract and concrete class
It can implement an interface that contains any number of abstract methods	It can implement an interface which contains a single abstract methods
Inside this we can declare instance variables	It does not allow declaration of instance variables, whether the variables declared simply act as local variables
Anonymous inner class can be instantiated	Lambda expression can not be instantiated
Inside Anonymous inner class, "this" always refers to current anonymous inner class object but not to outer object	Inside Lambda expression, "this" always refers to current outer class object that is, enclosing class object
It is the best choice if we want to handle multiple methods	It is the best choice if we want to handle interface
At the time of compilation, a separate .class file will be generated	At the time of compilation, no separate .class file will be generated. It simply convert it into private method outer class
Memory allocation is on demand, whenever we are creating an object	It resides in a permanent memory of JVM

ect Run Window Help

Anonymous_Inner_Class.java | Anonymous_Inner_Class_with_lambda_expression.java

```
1  package _03_Anonymous_Inner_Classes;
2
3  interface Interf
4  {
5     public void m1();
6  }
7  public class Acessing_Inner_And_Outer_Variable {
8
9     public void m2() {
10       int x = 10;
11       Interf interf = () -> {
12             int y=20;
13             System.out.println("x : "+x);
14             System.out.println("y : "+y);
15       };
16       interf.m1();
17    }
18    public static void main(String[] args) {
19       Acessing_Inner_And_Outer_Variable var = new Acessing_Inner_And_Outer_Variable();
20       var.m2();
21    }
22
```

Problems | Javadoc | Declaration | Console | SonarLi

<terminated> Acessing_Inner_And_Outer_Variable [Java Applicatic

```
x : 10
y : 20
```

```
iable.java
ambda_expres
```

```
 2
 3 interface Interf
 4 {
 5    public void m1();
 6 }
 7 public class Acessing_Inner_And_Outer_Variable {
 8    int x=10;
 9    public void m2() {
10       int y = 20;
11       Interf interf = () -> {
12
13             System.out.println("x : "+x);
14             System.out.println("y : "+y);
15             x=888;
16             y=11; // we can't change local variable which are referenced inside the lambda expression,
17       };
18       interf.m1();
19    }
20    public static void main(String[] args) {
21       Acessing_Inner_And_Outer_Variable var = new Acessing_Inner_And_Outer_Variable();
22       var.m2();
23    }
```

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.

```
default void m1() {
    System.out.println("Default Method");
}
```

➔ 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.

Default Methods in Interfaces Example:

```
interface ExampleInterface {
    default void m1() {
        System.out.println("Default Method");
    }
}

class ExampleClass implements ExampleInterface {
    public static void main(String[] args) {
        ExampleClass example = new ExampleClass();
        example.m1();
    }
}
```

➔ 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.

Example:

```
interface InvalidInterface {
    default int hashCode() {
        return 10;
    }
}
```

Compile-Time Error: The reason is that Object class methods are by default available to every Java class, so it's not required to bring them through default methods.

For further detailed information, please visit: Default Methods in Interfaces in Java 8 — Examples

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.

```java
// Interface A
interface A {
    void show();
}

// Interface B
interface B {
    void show();
}

// Class implementing both interfaces
class C implements A, B {
    // Class must provide its own implementation for the show() method
    public void show() {
        System.out.println("C's show()");
    }
}

public class DiamondProblemExample {
    public static void main(String[] args) {
        C c = new C();
        c.show();  // Output: C's show()
    }
}
```

```java
interface A {
    default void show() {
        System.out.println("A's show()");
    }
}

interface B {
    default void show() {
        System.out.println("B's show()");
    }
}

class C implements A, B {
    // The diamond problem occurs here, need to override the method
    @Override
    public void show() {
        A.super.show();  // Resolving the conflict by calling A's show()
    }
}

public class DiamondProblemExample {
    public static void main(String[] args) {
        C c = new C();
        c.show();  // Output: A's show()
    }
}

```

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.

```java
public interface Medium {

    // static methods inside the interface must have a body.
    // Static methods also cannot be abstract.
    static void myStaticMethod(){
        System.out.println("This is a static method");
    }

    public static void main(String[] args) {
        Medium.myStaticMethod();
        // We can not create an object from interface
        // But we can access to static method with interface's name (Medium)
    }

}
```

```java
interface A {
    public static void m1(){
      System.out.println("call me with the Interface name");
    }
}

class Test implements A {
    public static void main(String[] args){
      A.m1(); //only way to call interface's static methods
    }
}
```

Can we Override Interface’s static methods?

As aforementioned, Interface’s static methods can’t be overridden in implementing classes or in extending interfaces as they are not available to child classes by default. However, we can declare methods with the same signature in the implementing classes. They are valid, but we will not consider them as overridden methods. Hence, in this scenario method overriding concept is not valid. Also from the previous versions of Java, we know that static methods don’t participate in overriding.

```
interface A {
    public static void m1(){
      System.out.println("I am Interface's static method");
    }
}

class Test1 implements A {
    public static void m1(){
      System.out.println("I am not an overridden method");
    }
}

class Test2 implements A {
    public void m1(){
      System.out.println("I am not an overridden method");
    }
}

class Test3 implements A {
    private static void m1(){
      System.out.println("I am not an overridden method");
    }
}
```

Predefined Functional Interface

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

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

Predicate

PredicateExample.java

```
/**
 *    ***************************** Predicate ********************
 *     **) Introduced in 1.8 V
 *     **) Present in java.util.function package.
 *     **) method : test()
 *     **) prototype signature
 *
 *          interface Predicate<T>
 *          {
 *            boolean test(T t);
 *          }
 *
 */
public class PredicateExample {

  public static void main(String[] args) {
    /**
     *  Check the given number is greater than 10 or not.
     *
     *  **) As Predicate is default given by Java people , we can directly reference lambda to it.
     */
     Predicate<Integer> predicate = (number) -> number >10 ;
     boolean response = predicate.test(11);
     System.out.println(response);
```

Pro... Jav... Dec... Con... So...

<terminated> PredicateExample [Java Application]

```
true
```

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.

```
/**
 *    ********************* Predicate Joining ***************************
 *    **) Multiple predicate join together to check multiple condition together.
 *    **) Different  methods which help to achieve this functionality
 *
 *          --) and()       : Both predicate condition must be true , then only true.
 *          --) or()        : At least one of the condition should be true , then it will be true
 *          --) negate()    : It will check opposite logic of predicate provided.
 *
 *    **)  P1 : Given number is greater than 10 or not.
 *    **) P2 : Given number is Even or not.
 *
 *    **) P3 : P1.and(P2)  : This will check for join predicate condition .
 *
```

```
 *
 */
public class Predicate_Joining {
  /**
   *
   *  **) I have to find whether a number is greater than 10 or not and it is even together.
   *  **)
   */
  public static void main(String[] args)
  {
    Predicate<Integer> evenNumberCheck = (number)-> number%2==0;
    Predicate<Integer>  greaterThanTenCheck = (number)->number > 10 ;

    Predicate<Integer> joinPredicateWithAnd = evenNumberCheck.and(greaterThanTenCheck);
    Predicate<Integer> joinPredicateWithOr = evenNumberCheck.or(greaterThanTenCheck);
    Predicate<Integer> isNotGreaterThanTen = greaterThanTenCheck.negate();


    System.out.println("and condition check  : " +joinPredicateWithAnd.test(12));
    System.out.println("Or condition check : "+joinPredicateWithOr.test(11));
    System.out.println("negation check : "+isNotGreaterThanTen.test(12));

```

PredicateExample.java | Predicate_Joining.java | Predicate.class | Employee_Filtering_Using_Predicate.java ×

```
54 public class Employee_Filtering_Using_Predicate {
55   public static void main(String[] args) {
56     Predicate<Employee> salaryGreaterThan20000 = (emp) -> emp.getEmpSalary() > 20000;
57     Predicate<Employee> employeeWhoseDesignationIsCEO =
58                   (emp) -> emp.getDesignation().equalsIgnoreCase("CEO");
59
60     ArrayList<Employee> listEmployee = new ArrayList<>();
61     listEmployee.add(new Employee(1, "Karan", 21000, "CEO"));
62     listEmployee.add(new Employee(2, "Mahi", 23000, "MANAGER"));
63     listEmployee.add(new Employee(3, "Monika", 19000, "DEVELOPER"));
64     listEmployee.add(new Employee(4, "Soumya", 18000, "CEO"));
65     listEmployee.add(new Employee(5, "Amit", 22000, "MANAGER"));
66     listEmployee.add(new Employee(6, "Akshat", 81000, "TESTER"));
67
68     System.out.println("Employee having salary greater than 20000");
69     for (Employee employee : listEmployee) {
70       if (salaryGreaterThan20000.test(employee)) {
71         System.out.println(employee);
72       }
73     }
74
75     System.out.println("Employee having designation CEO");
76     for (Employee employee : listEmployee) {
77       if (employeeWhoseDesignationIsCEO.test(employee)) {
78         System.out.println(employee);
```

Function

➔ when our input type can be anything and output type can be anything then we can go with **Function** Functional Interface.

Employee_Filtering_Using_Predicate.java | FunctionExample.java

```
6  *     ***************************** Function **********
7  *      **) Introduced in 1.8 V
8  *      **) Present in java.util.function package.
9  *      **) method : apply()
10 *      **) prototype signature
11 *
12 *            interface Function<T,R>
13 *            {
14 *              R apply(T t);
15 *            }
16 *
17 */
18 public class FunctionExample {
19
20   public static void main(String[] args) {
21     /**
22      *  Find length of Given String
23      *
24      *  **) As Function is default given by Java people , we can directly reference lambda to it.
25      */
26     Function<String,Integer> function = (string) ->string.length();
27     int lengthOfString = function.apply("Karan");
28     System.out.println(lengthOfString);
```

Problems | Javadoc | Declaration | Console | Sonar

<terminated> FunctionExample [Java Application] C:\Projects\st

5

Find_Grade_Of_Student.java × FunctionExample.java

```
50  {
51      /**
52       *  Find Grade of Each Student
53       */
54      Function<Student, Character> gradeFunction = (student) -> student.getStudentMarks() >= 80 ? 'A'
55              : student.getStudentMarks() >= 60 && student.getStudentMarks() < 80 ? 'B'
56                      : student.getStudentMarks() >= 50 && student.getStudentMarks() < 60 ? 'C'
57                              : student.getStudentMarks() > 40 && student.getStudentMarks() < 50 ? 'D' : 'E';
58
59      List<Student> list = new ArrayList<Student>();
60      list.add(new Student(101,"Ram",95));
61      list.add(new Student(102,"Shyam",35));
62      list.add(new Student(103,"Karan",75));
63      list.add(new Student(104,"Vishal",85));
64      list.add(new Student(105,"Chiku",65));
65      list.add(new Student(106,"Mota",54));
66      list.add(new Student(107,"Mahi",45));
67
68      System.out.println("Name and Grade of Student");
69      for(Student student : list) {
70          System.out.println("Name : "+student.getStudentName() +
71                  " Grade :  "+gradeFunction.apply(student));
72      }
```

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.**

Find_Grade_Of_Student.java | FunctionExample.java | Function_Chaining.java | Problems | Javadoc | Declaration | Console | SonarLint On-Th

<terminated> Function_Chaining [Java Application] C:\Projects\sts-4.19.0.

```
7
Square of 4 is :
16
Square of a string length of Anamika
49
```

```java
 *
 */
public class Function_Chaining {

    public static void main(String[] args)
    {
      Function<String,Integer> lengthOfString = (string) -> string.length();
      Function<Integer,Integer> squareOfNumber = (number) -> number* number;

      System.out.println("Length of the string Anamika : ");
      System.out.println(lengthOfString.apply("Anamika"));

      System.out.println("Square of 4 is : ");
      System.out.println(squareOfNumber.apply(4));

      /**
       *  Chaining both condition together
       */
      System.out.println("Square of a string length of Anamika");
      System.out.println(lengthOfString.andThen(squareOfNumber).apply("Anamika"));

    }
```

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.

*ConsumerExample.java

```
 8 *     ***************************** Consumer ***************
 9 *      **) Introduced in 1.8 V
10 *      **) Present in java.util.function package.
11 *      **) method : accept()
12 *      **) prototype signature
13 *
14 *            interface Consumer<T>
15 *            {
16 *              void accept(T t);
17 *            }
18 *
19 */
20 public class ConsumerExample {
21   public static void main(String[] args) {
22         List<String> cities = new ArrayList<>();
23         cities.add("Delhi");
24         cities.add("Mumbai");
25         cities.add("Goa");
26         cities.add("Pune");
27
28         Consumer<String> printConsumer= city-> System.out.println(city);
29         cities.forEach(printConsumer);
30   }
31
32 }
```

Probl... Java... Decla... Cons... Sona...

<terminated> ConsumerExample [Java Application] C:\Proj

```
Delhi
Mumbai
Goa
Pune
```

Methods :

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

Definition:

The `Consumer.andThen()` method provides a way to compose two instances of `Consumer`, effectively chaining them together. After performing its operation, the first `Consumer` will pass its input to the second `Consumer`.

Syntax:

```
default Consumer<T> andThen(Consumer<? super T> after)
```

Key Points:

- The `andThen()` method allows for the sequential combination of multiple `Consumer` operations.

- If the `andThen()` operation's argument throws an exception, it is relayed to the caller.

- If the original `Consumer` (the one calling `andThen()`) throws an exception, the `after` `Consumer` will not be executed

```
    public static void main(String[] args) {
        Consumer<String> toUpperCase = s -> System.out.println(s.toUpperCase());
        Consumer<String> appendExclamation = s -> System.out.println(s + "!");

        // Chaining the two consumers
        Consumer<String> chainedConsumer = toUpperCase.andThen(appendExclamation);

        String input = "hello";
        chainedConsumer.accept(input);
    }
}
```

Output:

```
HELLO
hello!
```

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.

- ✅ **T (Return Type):** The type of value that the `Supplier` provides.
- ✅ **Common Use Cases:**
  - ✔ **Generating dynamic values** – Timestamps, random numbers, unique IDs.
  - ✔ **Lazy initialization** – Creating objects only when needed.
  - ✔ **Providing default configurations** – Supplying fallback values when required.

✔ Example: Returning a Constant Value

```java
import java.util.function.Supplier;

public class SupplierGetExample {
    public static void main(String[] args) {
        // ✅ Define a Supplier that returns a constant value
        Supplier<String> constantSupplier = () -> "Hello World!";

        String result = constantSupplier.get();
        System.out.println(result); // Output: Hello World!
    }
}
```

ConsumerExample.java | Consumer_Chaining.java | SupplierExample.java | Probl... | Java... | Decla... | Cons... | Sc

<terminated> SupplierExample [Java Application] C:

```
Hello World!
```

```
/**
 *     ****************************** Supplier ***
 *      **) Introduced in 1.8 V
 *      **) Present in java.util.function package.
 *      **) method : get()
 *      **) prototype signature
 *
 *            interface Supplier<R>
 *            {
 *              R get();
 *            }
 *
 */
public class SupplierExample {
      public static void main(String[] args) {
            // ☑ Define a Supplier that returns a constant value
            Supplier<String> constantSupplier = () -> "Hello World!";

            String result = constantSupplier.get();
            System.out.println(result); // Output: Hello World!
        }

```

📌 **Why use `Supplier` here?**

✅ **Encapsulates logic inside a function.**

✅ **Provides flexibility to return values dynamically.**

🚀 **Use `Supplier<T>` for functions that return values without input!**

2️⃣ Generating Dynamic Data with `Supplier`

✔️ Example: Returning the Current Date and Time

```
import java.time.LocalDateTime;
import java.util.function.Supplier;

public class SupplierGetExample {
    public static void main(String[] args) {
        // ✅ Supplier that returns the current date and time
        Supplier<LocalDateTime> dateTimeSupplier = () -> LocalDateTime.now();

        LocalDateTime currentDateTime = dateTimeSupplier.get();
        System.out.println(currentDateTime);
```

3️⃣ Using `Supplier` for Default Configurations

✔ Example: Providing a Default Configuration

```
import java.util.function.Supplier;

class Configuration {
    private String url;
    private int timeout;

    public Configuration(String url, int timeout) {
        this.url = url;
        this.timeout = timeout;
    }
```

```
    }
}

public class DefaultConfigurationSupplierExample {
    public static void main(String[] args) {
        // ✅ Supplier that provides a default configuration
        Supplier<Configuration> defaultConfiguration =
            () -> new Configuration("http://localhost:8080", 50000);

        Configuration configuration = defaultConfiguration.get();
        System.out.println(configuration.toString());
    }
}
```

✔ Use Case 2: Generating Random OTP (One-Time Password)

```
import java.util.function.Supplier;
import java.util.Random;

public class OTPGenerator {
    public static void main(String[] args) {
        // ✅ Supplier to generate a 6-digit OTP
        Supplier<String> otpSupplier = () -> {
            Random random = new Random();
            return String.format("%06d", random.nextInt(1000000));
        };

        System.out.println("Generated OTP: " + otpSupplier.get());
    }
}
```

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

24) Can we consider streams as another type of data structure in Java? Justify your answer?

You can't consider streams as data structure. Because they don't store the data. You can't add or remove elements from the streams. They are the just operations on data. Stream consumes a data source, performs operations on it and produces the result. Source may be a collection or an array or an I/O resource. They don't modify the source.

25) What are intermediate and terminal operations?

The operations which return stream themselves are called intermediate operations. For example – `filter()`, `distinct()`, `sorted()` etc.

The operations which return other than stream are called terminal operations. `count()`. `min()`, `max()` are some terminal operations.

29) What are short circuiting operations?

Short circuiting operations are the operations which don't need the whole stream to be processed to produce a result. For example – `findFirst()`, `findAny()`, `limit()` etc.

40) Name any 5 methods of Collectors class and their usage?

Method	Description
joining()	Concatenates input elements separated by the specified delimiter.
counting()	Counts number of input elements
groupingBy()	Groups the input elements according supplied classifier and returns the results in a *Map*.
partitioningBy()	Partitions the input elements according to supplied *Predicate* and returns a *Map<Boolean, List<T>>*
toList()	Collects all input elements into a new *List*

45) Name three important classes of Java 8 Date and Time API?

`java.time.LocalDate`, `java.time.LocalTime` and `java.time.LocalDateTime`

46) How do you get current date and time using Java 8 features?

```
1 LocalDateTime currentDateTime = LocalDateTime.now();
```

Functional Interfaces having 2 input arguments

BiPredicate

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

test : It is an abstract method

Other methods are default method.

This interface represents a predicate that takes two arguments.

This how the interface is defined:

```
@FunctionalInterface
public interface BiPredicate<T, U> {
     boolean test(T t, U u);
     // Default methods are defined also
}
```

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

        // anonymous class implements BiPredicate interface
        BiPredicate < String, String > predicateObject = new BiPredicate < String, String > () {

            @Override
            public boolean test(String s1, String s2) {
                return s1.equals(s2);
            }
        };
        System.out.println(predicateObject.test("Ramesh", "Ramesh"));

        // Lambda expression implementation
        BiPredicate < String, String > biPredicate = (s1, s2) -> s1.equals(s2);
        System.out.println(biPredicate.test("ramesh", "ramesh"));
        System.out.println(biPredicate.test("java guides", "Java Guides"));
    }
```

Java 8 - BiPredicate.and(), BiPredicate.or() Method Example #2

```
package com.javaguides.java.functionalinterfaces;

import java.util.function.BiPredicate;

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

        // Demo for BiPredicate.and() and BiPredicate.or() methods
        BiPredicate < Integer, Integer > biPredicate2 = (x, y) -> x > y;
        BiPredicate < Integer, Integer > biPredicate3 = (x, y) -> x == y;

        // usage of and() method - and condition
        boolean result = biPredicate2.and(biPredicate3).test(20, 10);
        System.out.println(result);
```

BiFunction

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

🚀 Introduction to Java `BiFunction` Functional Interface

In Java **functional programming**, the `BiFunction<T, U, R>` interface (from `java.util.function`) is a **functional interface that takes two input arguments and returns a result**.

- ✅ **T (First Input Type):** The type of the first argument.
- ✅ **U (Second Input Type):** The type of the second argument.
- ✅ **R (Return Type):** The type of the result.

💡 Common Use Cases:

- ✔ **Performing calculations**—Addition, subtraction, multiplication, and division.
- ✔ **Processing two inputs**—Combining two values into one result.
- ✔ **Transforming and mapping values**—Applying a transformation based on two inputs.

📌 In this article, you'll learn:

- ✅ How to **use `BiFunction<T, U, R>` with examples**.
- ✅ How to use **`apply()`** to process values.
- ✅ How to **chain `BiFunction` with `Function` using `andThen()`**

BiPredicateExample.java | BiFunctionExample.java ×

```
{
    /**
     * I wanted to find a student whose age is greater than 18 and have voter id can vote
     */
    List<Student> listStudent = new ArrayList<>();
    listStudent.add(new Student(101,"Karan",true,29));
    listStudent.add(new Student(102,"Rahul",true,27));
    listStudent.add(new Student(103,"Mahi",true,29));
    listStudent.add(new Student(104,"Anurag",false,29));
    listStudent.add(new Student(105,"Amit",true,29));
    listStudent.add(new Student(106,"Nihar",false,16));
    listStudent.add(new Student(107,"Akshat",true,17));
    listStudent.add(new Student(108,"Elango",false,39));

    BiFunction<Integer,Boolean,Boolean> ageAndVoterIdCheckPredicate = (age, voterIdCheck) -> (age > 18 && voterIdCheck ==true) ? true : fa
    System.out.println("Student Name who is eligible for vote");
    listStudent.forEach(student->{
        if(ageAndVoterIdCheckPredicate.apply(student.getAge(), student.isVoterId())){
            System.out.println("Name : "+student.getStudentName() + ", age : "+student.getAge());
        }
    });
}
```

Probl... | Java... | Decla... | Cons... × | Sona...

<terminated> BiFunctionExample [Java Application] C:\Pro

```
Student Name who is eligible for vote
Name : Karan, age : 29
Name : Rahul, age : 27
Name : Mahi, age : 29
Name : Amit, age : 29
```

Primitive Functional Interface for Predicate

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

Normal Functional Interface

Primitive Functional Interface

Finding Even Number using Predicate

Int arr[] = {1,2,3,4,5,6,7,8};

Predicate<Integer> predicate = (i) ->i%2==0

This will take Object but

we are checking for

Primitive type from arr. So,

Int ⟶ Integer ➡ then for operation it again needs to be

Converted to primitive int

So for N number of input , N number of times

These conversion required.

Hence it is worst type of implementation from Predicate.

IntPredicate intPredicate = (i)-> i%2==0;

Now from IntPredicate It takes primitive int

as input , so conversion is not required.

So it is best implementation than Predicate.

PredicateExample.java

```java
package 09_Primitive_Type_Functional_Interface;

import java.util.Arrays;
import java.util.function.IntPredicate;
import java.util.function.Predicate;

public class PredicateExample {

    public static void main(String[] args) {

        /**
         * Find the even number from the arr
         */
        int[] arr =  {1,2,3,4,5,6,7,8};
        Predicate<Integer> predicate = (i)-> i%2==0; // Normal Functional Interface
        IntPredicate biPredicate = (i) ->i%2==0;   // Primitive Functional Interface

        System.out.println("Printing even through Predicate");
        Arrays.stream(arr).filter(number->predicate.test(number)).forEach(System.out::println);
        System.out.println("Printing even through IntPredicate");
        Arrays.stream(arr).filter(number->biPredicate.test(number)).forEach(System.out::println);
```

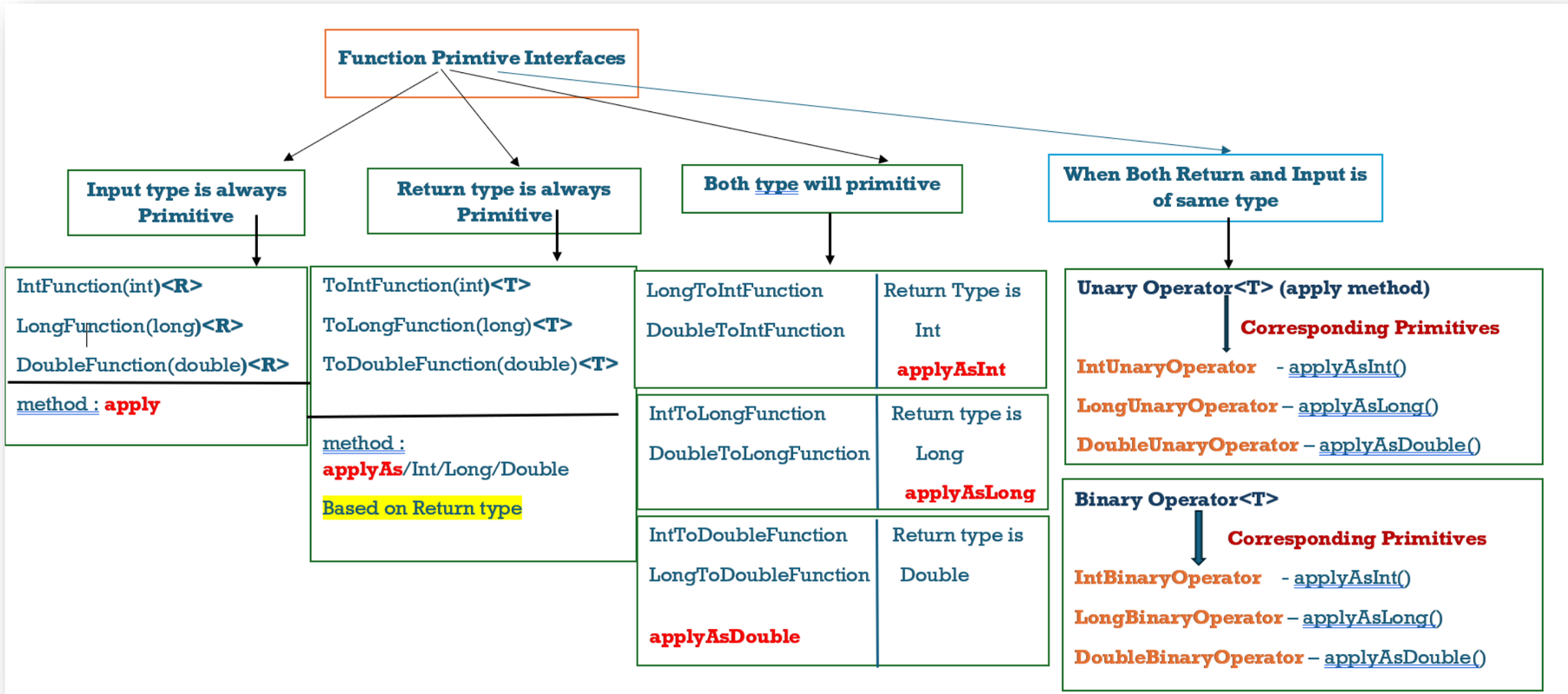
Problems Javadoc Declaration Console SonarLint On..

<terminated> PredicateExample (1) [Java Application] C:\Projects\sts-4.

```
Printing even through Predicate
2
4
6
8
Printing even through IntPredicate
2
4
6
8
```

Primtive Functional Interfaces for Function

Different Primtive Type of Functional Functional Interface

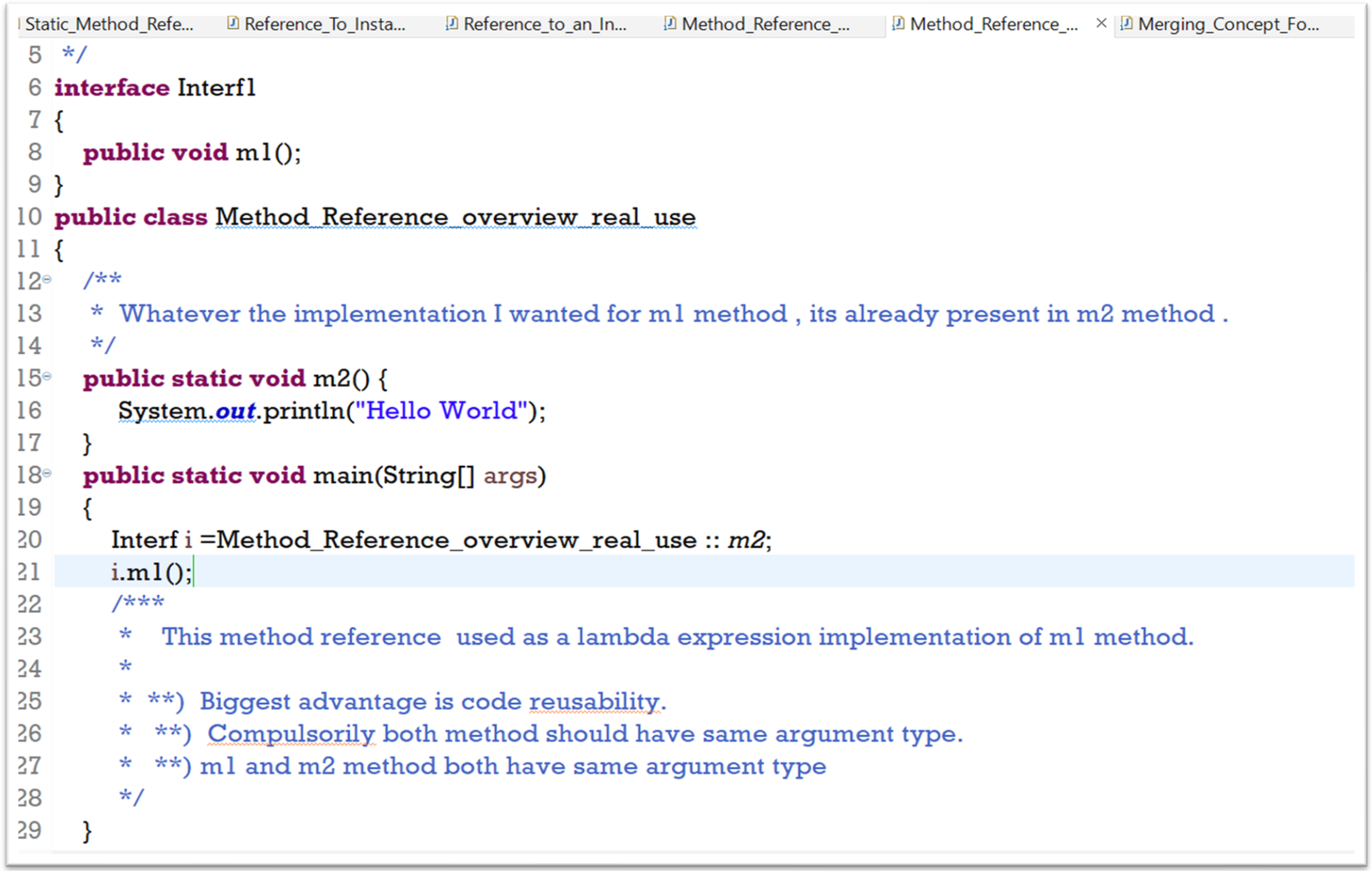


UnaryOperator → One Input one Output and both of same type .
BinaryOperator → Two Input and one Output all are of same type.

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.

```
interface Interf
{
   public void m1();
}
public class Method_Reference_overview
{
   public static void main(String[] args)
   {
      Interf i =()-> System.out.println("Hello World"); // This is the Lambda expression implementation for
      i.m1();                                           // m1 method

   }

}
```



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

Method_Reference.java

```java
package _10_method_and_constructor_reference;

import java.util.Arrays;
public class Method_Reference {

   public static void uppercase(String s) {
       System.out.println(new String(s).toUpperCase());
   }
   public static void main(String[] args)
   {
      String[] arr = {"hello","how","are","you"};
      /**
       *  ********** Static Method Reference *****************
       *    **) Class method should be static.
       *    **) Can be used in place of lambda expression.
       *
       *
       */
      /**   This is normal approach   */
      Arrays.stream(arr).forEach(word->Method_Reference.uppercase(word));

      /** This is static Method Reference approach*/
      Arrays.stream(arr).forEach(Method_Reference::uppercase);

   }
```

Instance Method Reference

```
49
50 public class Reference_To_Instance_method_Particular_Object {
51   public static void main(String[] args)
52   {
53     BicycleComparator bikeFrameSizeComparator = new BicycleComparator();
54     List<Bicycle> createBicyclesList = new ArrayList<>();
55     createBicyclesList.add(new Bicycle("Hero",5));
56     createBicyclesList.add(new Bicycle("Honda",6));
57     createBicyclesList.add(new Bicycle("Nexas",15));
58     createBicyclesList.add(new Bicycle("Auto",3));
59     createBicyclesList.add(new Bicycle("Mercedes",25));
60     createBicyclesList.add(new Bicycle("Audi",35));
61
62     /** Normal Way*/
63     createBicyclesList.stream().sorted((a,b)->bikeFrameSizeComparator.compare(a, b));
64     /** With Instance Method Reference */
65     createBicyclesList.stream().sorted(bikeFrameSizeComparator::compare);
66
67     System.out.println(createBicyclesList);
68
69 }
```

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

Static_Method_Reference.java | Reference_To_Instance_method_Particular_Object.java | Reference_to_an_Instance_Method_of_an_Arbitrary_Object_of...

```java
package _10_method_and_constructor_reference;

import java.util.Arrays;
import java.util.List;

public class Reference_to_an_Instance_Method_of_an_Arbitrary_Object_of_a_Particular_Type
{
   public static void main(String[] args)
   {
      List<Integer> numbers = Arrays.asList(5, 3, 50, 24, 40, 2, 9, 18);

      /***
       * Normal Way
       */
      numbers.stream()
        .sorted((a, b) -> a.compareTo(b));

      /***
       *   Reference_to_an_Instance_Method_of_an_Arbitrary_Object_of_a_Particular_Type  Way
       */
      numbers.stream()
        .sorted(Integer::compareTo);

      System.out.println(numbers);

```

Constructor Reference

```
interface Interf4
{
   public Sample get();
}
class Sample
{
   Sample()
   {
      System.out.println("Hello from  Sample Constructor");
   }
}
public class Constructor_Reference {

   public static void main(String[] args)
   {
    Interf4 interf = ()->{
          Sample s= new Sample();
          return s;
    };
    Sample sample = interf.get();
    System.out.println(sample);

   }
```

```
3
4  public class Constructor_Reference_Refactored {
5
6      public static void main(String[] args)
7      {
8          Interf4 interf = Sample::new;
9          Sample sample = interf.get();
10         /**
11          * Referring sample class constructor implementation to interface4 get method.
12          * Get method internally refers sample class new  args constructor
13          */
14         System.out.println(sample);
15
16     }
17
18 }
```

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.

Some Method includes

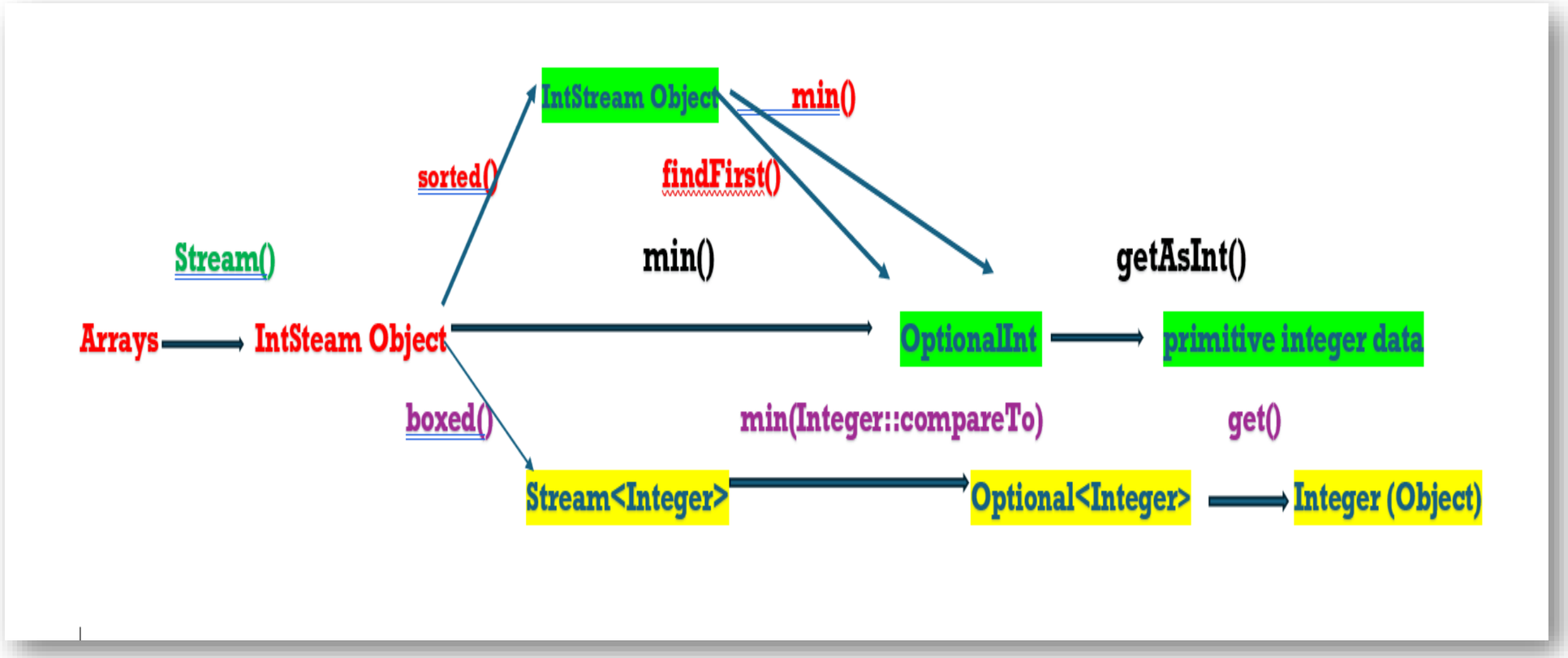
anyMatch()	findFirst()
min()	forEach()
allMatch()	max()
noneMatch()	findAny()
collect()	reduce()
count()	toArray()

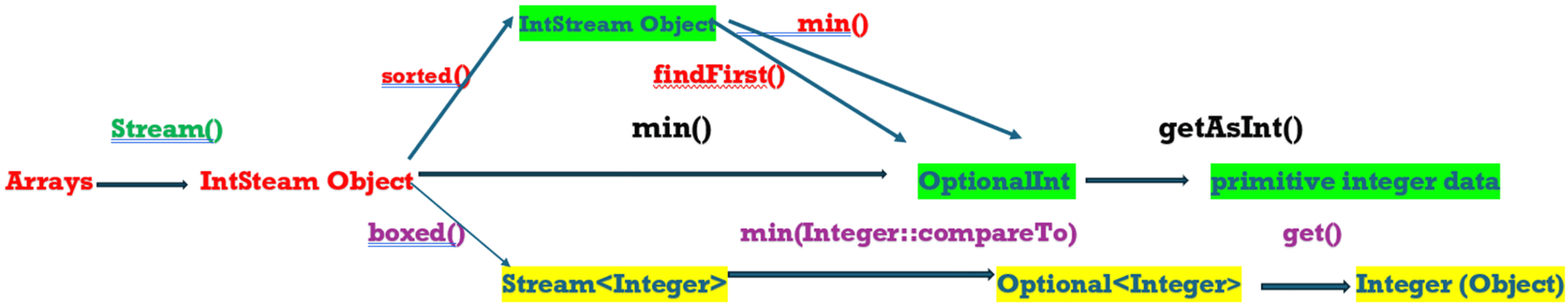
➔ Java 8 Streams support **intermediate operations** that **return another stream**.
➔ Allowing developers to chain multiple operation together to form the **pipeline**.
➔ Intermediate operations are lazy, meaning they are not executed until a terminal operation is invoked.

Some Method includes

1. filter()
2. map()
3. flatMap()
4. distinct()
5. sorted()
6. peek()
7. limit()
8. skip()

Sorting


IntStream Object
min()
sorted()
findFirst()
Stream()
min()
getAsInt()
Arrays
IntSteam Object
OptionalInt
primitive integer data
boxed()
min(Integer::compareTo)
get()
Stream<Integer>
Optional<Integer>
Integer (Object)

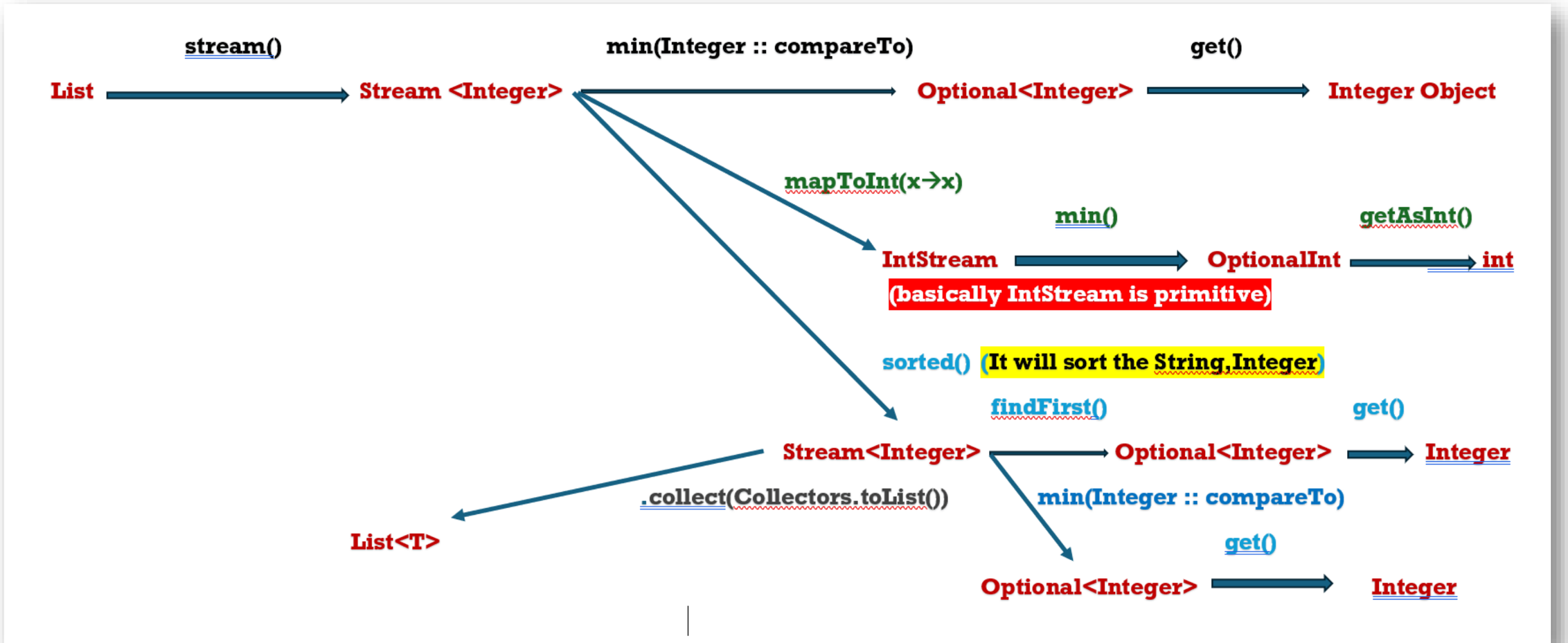


Note :

**) min() method we can apply both on primitive int and Object Integer.

**) on primitive int , It does not required anything as method argument and it will return OptionalInt

**) Whereas Object Integer require Comparator object and return Optional<Integer>.


stream()
min(Integer :: compareTo)
get()
List
Stream <Integer>
Optional<Integer>
Integer Object
mapToInt(x→x)
min()
getAsInt()
IntStream
OptionalInt
int
(basically IntStream is primitive)
sorted()
(It will sort the String,Integer)
findFirst()
get()
Stream<Integer>
Optional<Integer>
Integer
.collect(Collectors.toList())
min(Integer :: compareTo)
get()
List<T>
Optional<Integer>
Integer

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://javaconceptoftheday.com/solving-real-time-queries-using-java-8-features-employee-management-system/#google_vignette

```
/**
 * Sort the Map Based on Descending order of Key
 */
List<Entry<String, Integer>> sortingbasedOnKeyDesc = nameToSalMapping.entrySet().stream()
        .sorted(Comparator.comparing(Map.Entry::getKey, Comparator.reverseOrder()))
        .collect(Collectors.toList());
```

```
List<Employee> sortedListbasedOnEmpNameReverseOrder = empList.stream()
        .sorted(Comparator.comparing(Employee::getEmpName).reversed()).collect(Collectors.toList());
System.out.println("sortedListbasedOnEmpNameReverseOrder : " + sortedListbasedOnEmpNameReverseOrder);
```

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