1. Functional Interface
2. Predicate
3. Function
4. Method references
5. Stream API

**Course Content**

1. Functional Interface
   1. Different ways of implementation
   2. Predefined functional interface with real time use cases
2. Lambda Expression
   1. Rules
   2. Real time examples
3. Predicate
4. Function
5. Method references
6. Optional class
7. Stream API
   1. Creating stream
   2. Intermediate operations
   3. Terminal operations
   4. Realtime examples
8. Debugging skill
9. GIT tool real time example

------------------------------------------------------------------------------------------------------------------------------------------

# Functional Interface / Single Abstract Method Interface(SAM)

What is interface

Difference between interface and abstract class

How many ways we can implement interface (impl class + annonymous class)

Definition of functional interface

Can we override functional interface?

Functional interface with inheritance

Predefined Functional interfaces with example

Rules:

1) We write any number of **default and static methods.**

2) **@FunctionalInterface** is required to write but it should have only one abstract method

*Functional interface with inheritance*

* *If super interface is Functional, then child will be by default even if it does not contain any abstract method*
* *In the child interface we can define exactly same parent interface abstract method.*
* *In the child interface we can’t define any new abstract methods otherwise child interface won’t be FunctionalInterface and if we are trying to use @FunctionalInterface annotation then compiler gives an error message.*

# Lambda Expression

2 Way to implement interface using impl class + Anonymous class

Why we are using lambda expression

How to write it (Rules)

***What are the advantages of Lambda expression?***

☀ We can reduce length of the code so that readability of the code will be improved.

☀ We can resolve complexity of anonymous inner classes.

☀ We can provide Lambda expression in the place of object.

☀ We can pass lambda expression as argument to methods. (behavioral parameter)

public static void main(String[] args) {

m1((str)->str.toUpperCase(),"ankur");

}

public static void m1(Function<String, String> myFunction,String ss) {

System.out.println(myFunction.apply(ss));

}

**Rules:**

1) A lambda expression **can have zero or more parameters**(arguments). Ex: ()  sop(“hello”); (int a )  sop(a); (inta, int b)  return a+b;

2) Usually we can **specify type of parameter.** If the compiler expects the type based on the context then we can remove type. i.e., programmer is not required. Ex: (inta, int b)  sop(a+b); (a,b)  sop(a+b);

3) If **multiple parameters present** then these parameters should be separated with comma(,).

4) If **zero number of parameters** available then we have to use empty parameter [ like ()]. Ex: ()  sop(“hello”);

5) If only one parameter is available and if the compiler can expect the type then we can remove the type and parenthesis also. Ex: (int a)  sop(a); (a) sop(a); A  sop(a);

6) Similar to method body lambda expression body also can contain **multiple statements**.if more than one statements present then we have to enclose inside within curly braces.if one statement present then curly braces are optional.

7) Once we write lambda expression we can call that expression just like a method, for this functional interfaces are required.

How we use it with functional interface

**Local variable and this keyword inside LE**

* Whatever the variables declare inside lambda expression are simply acts as local variables
* Within lambda expression ‘this” keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression)

**Default and Static method**

Difference between interface of java 8 and  7

Need of default and static method

What is default and static methods

Ways to call it from its subclass

Overridden rules for both methods

# Default methods

·         Default methods in interfaces help us to **introduce new functionality without breaking the contract of the implementing classes**.

·         Helps to **solve diamond problem** which occurs in multiple inheritance.

·         No compulsion to override the default method

·         We can override based on requirement.

**Static methods**

·         Used as a utility methods

·         Cannot override

# Predicate

·         A predicate is a function with a single argument and returns boolean value.

·         It’s a functional interface and it contains only one method i.e., **test()** , it accepts one argument and return Boolean value.

·         We can join multiple predicates by using and(), or() predicates also.

|  |  |
| --- | --- |
| Methods | Description |
| Boolean **test**(T t) | It evaluates this predicate on the given argument. |
| default Predicate<T>**and**(Predicate<? super T> other) | It returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. When evaluating the composed predicate, if this predicate is false, then the other predicate is not evaluated. |
| default Predicate<T>**negate**() | It returns a predicate that represents the logical negation of this predicate. |
| default Predicate<T>**or**(Predicate<? super T> other) | It returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. When evaluating the composed predicate, if this predicate is true, then the other predicate is not evaluated. |
| static <T> Predicate<T>**isEqual**(Object targetRef) | It returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object). |

# Function

* It is same as predicate only the difference is it accepts two argument and returns value of any type.
* It’s a functional interface and it contains only one method i.e., **apply()**

|  |  |
| --- | --- |
| **Method** | **Description** |
| * default <V> Function<T,V>**andThen**(Function<? super R,? extends V> after) | * It returns a composed function that first applies this function to its input, and then applies the after function to the result. If evaluation of either function throws an exception, it is relayed to the caller of the composed function. |
| * static <T> Function<T,T> identity() | * It returns a function that always returns its input argument. |
| * R **apply**(T t) | * It applies this function to the given argument. |
| * default <V> Function<V,R> compose(Function<? super V,? extends T> before) | * It Returns a composed function that first applies the before function to its input, and then applies this function to the result. If evaluation of either function throws an exception, it is relayed to the caller of the composed function. |

 Example:

Implementation class

public class **FunctionDemo** implements **Function<Employee, String>** {

public static void main(String[] args) {

Employee emp = new Employee(111, "Jiya Brein", 32, "Female", "HR", 2011, 25000.0);

System.out.println(new FunctionDemo().apply(emp));

}

@Override

public String apply(Employee t) {

return t.getDepartment();

}

}

Using anonymous class

public class FunctionDemo {

public static void main(String[] args) {

Employee emp = new Employee(111, "Jiya Brein", 32, "Female", "HR", 2011, 25000.0);

Function<Employee, String> function = new Function<Employee, String>() {

@Override

public String apply(Employee t) {

// TODO Auto-generated method stub

return t.getDepartment();

}

};

System.out.println(function.apply(emp));

}

}

Using Lambda expression

public class FunctionDemo {

public static void main(String[] args) {

Employee emp = new Employee(111, "Jiya Brein", 32, "Female", "HR", 2011, 25000.0);

Function<Employee, String> function = (emp1) -> emp.getDepartment();

System.out.println(function.apply(emp));

}

}

# Optional class

·         deal with NullPointerException in Java application

·         It provides methods which are used to check the presence of value for particular variable.

# Method reference

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

Java 8 method references can be defined as shortened versions of lambda expressions calling a specific method. Method references are the easiest way to refer a method than the lambdas calling a specific method. Method references will enhance the readability of your code.

## **Four types of method references**

1. Method reference to an instance method of an object – object::instanceMethod
2. Method reference to a static method of a class – Class::staticMethod
3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod
4. Method reference to a constructor – Class::new

Given example will show you the difference between lambda interface and method reference

@FunctionalInterface

**Public interface** PersonFunctionalInterface {

String getInstanceMethod(**int**age);

}

We can implement above interface using lambda expression as given below

**PersonFunctionalInterface person=(age)-> "Ankur's age is "+age;**

**System.out.println(person.getInstanceMethod(52));**

And same implementation you can do using method reference i. e. referring to interface method to instance or static method of class like below.

**publicclass**PersonDetails {

**public** String getAge(**int**age) {

**return**"Ankur's age is" + age;

}

}

**PersonFunctionalInterfacepersonI=new PersonDetails()::getAge;**

**System.out.println(personI.getInstanceMethod(5));**

Method reference is a shorthand notation of a lambda expression to call a method. For example:  
If your lambda expression is like this:

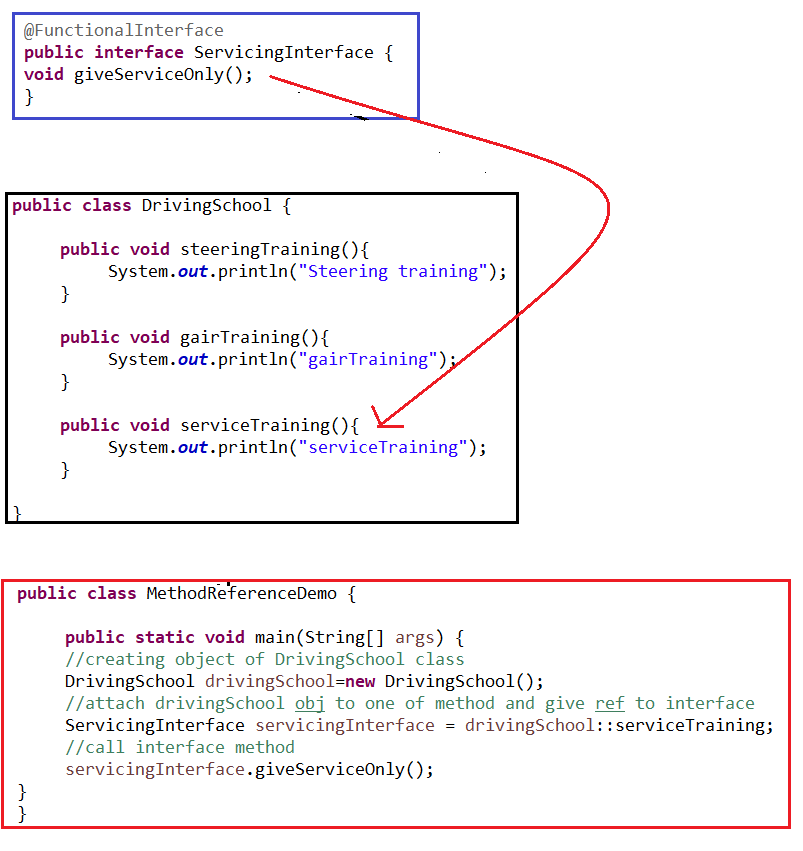
str->System.out.println(str)

then you can replace it with a method reference like this:

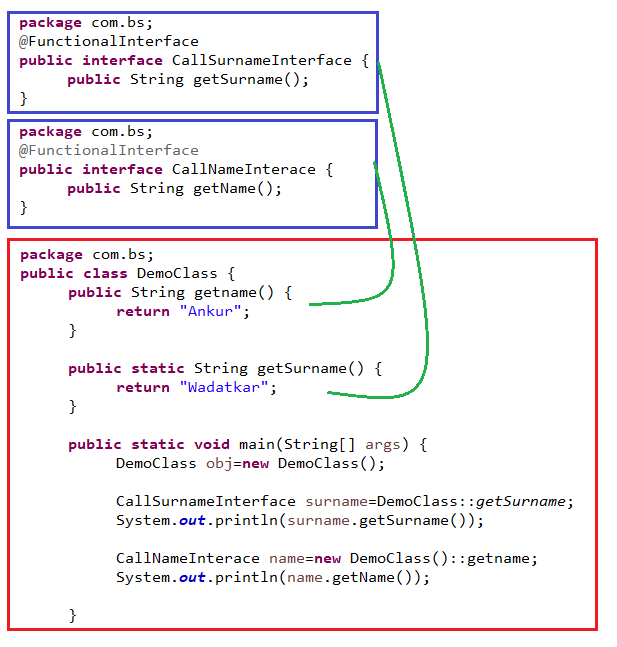
System.out::println

The :: operator is used in method reference to separate the class or object from the method name(we will learn this with the help of examples).

**1. Method reference to an instance method of an object – object::instanceMethod**

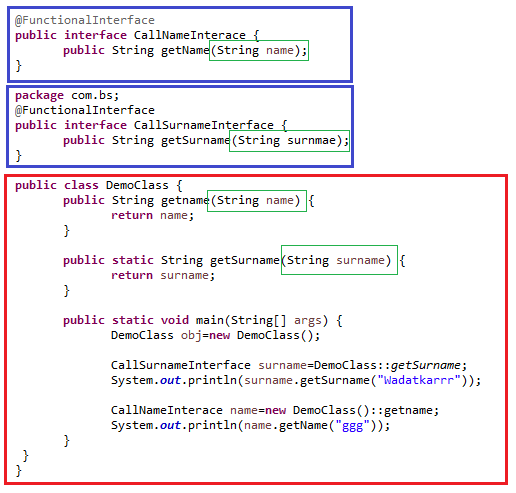


**2. Method reference to an static method of a class– ClassName::staticMethod**



**Method reference with arguments**

While referring to method no need to give an argument list, but when we will call interface method that time we should provide arguments. Note that the functional interface and class method both should have method arguments defined otherwise we may get compile time error. See below example.



Java 8 allows four types of method references.

|  |  |  |
| --- | --- | --- |
| **Method Reference** | **Description** | **Method reference example** |
| Reference to **static method** | Used to refer static methods from a class | **Math::max** equivalent to **Math.max(x,y)** |
| Reference to **instance method from instance** | Refer to an instance method using a reference to the supplied object | **System.out::println** equivalent to **System.out.println(x)** |
| Reference to **instance method from class type** | Invoke the instance method on a reference to an object supplied by the context | **String::length** equivalent to **str.length()** |
| Reference to **constructor** | Reference to a constructor | **ArrayList::new** equivalent to **new ArrayList()** |

## **Instance Method Reference of an Arbitrary Object**

When you refer to the instance method of a class with the ClassName , you will get the instance method reference of an arbitrary object of a particular type, such as ***ClassName::someInstanceMethod***;

Comparator<String>stringIgnoreCase = String::compareToIgnoreCase;

//this is equivalent to

Comparator<String>stringComparator = (first, second) ->first.compareToIgnoreCase(second);

**Optional class**

Why do we use it?

Several methods along with examples

Along with project also

<https://www.baeldung.com/eclipse-refactoring>

# Collection Sorting

We can use sort() of collection or we can use sorted() of java 8 to sort the collection object.

We can sort collection ascending by sort()/sorted() method but if we have to use custom sorting, we should use **Comparator.**

Comparator interface has a static method called **comparing** which accepts **Function.** In the function we have to return the value by which we have to sort our collection.

*The function we are using in comparing method can be implemented by* ***method reference or lambda expression.***

If we have to sort list of employee by firstName, then comparator will be

// using method reference

Comparator<Course>compareByFistName = Comparator.comparing(**Course::getFirstName**);

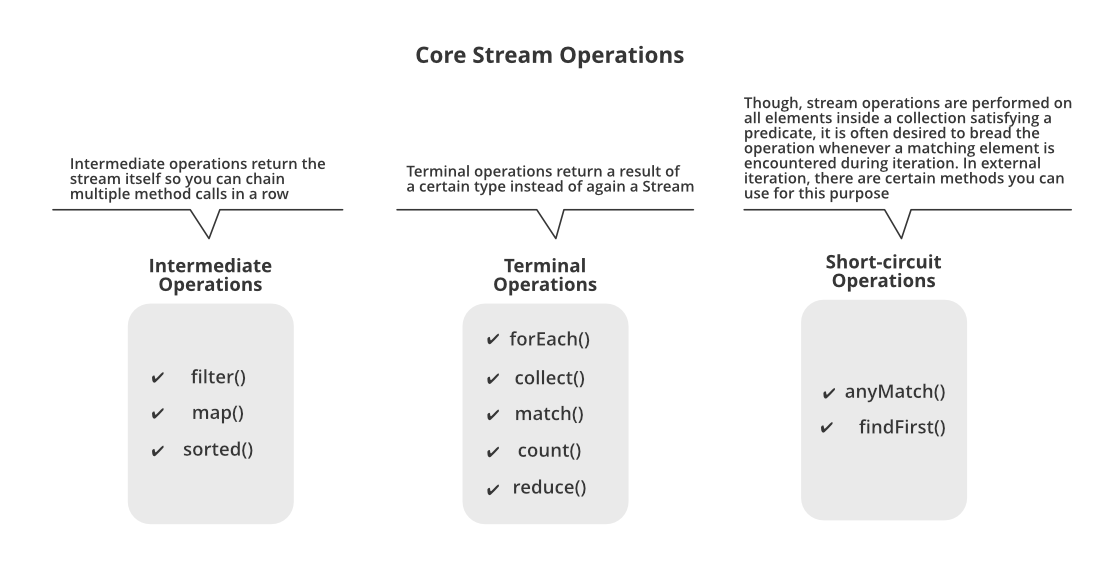
//using lambda ex

Comparator<Course>compareByFistName= Comparator.comparing(**object->object.getFirstName()**);

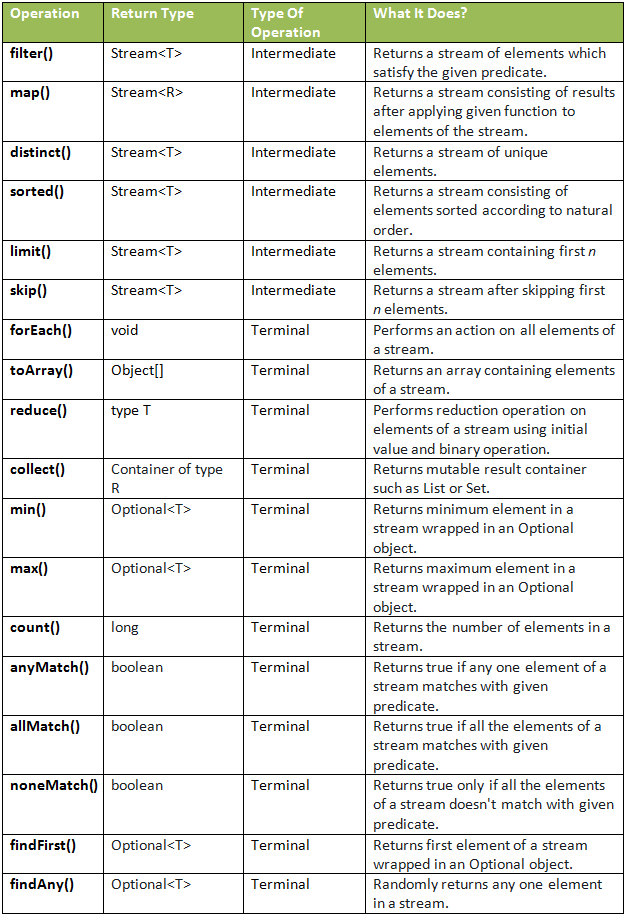
Collection sort mehod :<https://howtodoinjava.com/java/sort/collections-sort/>

# Stream API

1. Creation of stream
2. Intermediate operations
3. Terminal operation
4. Real time examples (main content)



Below is the list of intermediate and terminal operations.



Java 8 Stream Intermediate Vs Terminal Operations

|  |  |
| --- | --- |
| **Intermediate Operations** | **Terminal Operations** |
| They return stream. | They return non-stream values. |
| They can be chained together to form a pipeline of operations. | They can’t be chained together. |
| Pipeline of operations may contain any number of intermediate operations. | Pipeline of operations can have maximum one terminal operation, that too at the end. |
| Intermediate operations are lazily loaded. | Terminal operations are eagerly loaded. |
| They don’t produce end result. | They produce end result. |
| Examples : filter(), map(), distinct(), sorted(), limit(), skip() | Examples : forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny() |

**reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)->ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

Refer: <https://www.geeksforgeeks.org/stream-in-java/>

**Min() and Max()**

Integer[] intArr = { 1, 23, 2, 55, 65, 64, 26, 5, 56 };

System.out.println(Stream.of(intArr).min(***(o1, o2) -> o1 > o2 ? o2 : o1***).get());

System.out.println(Stream.of(intArr).max(**Integer::compareTo**).get());

System.out.println(Stream.of(intArr).max(**(a, b) ->a.compareTo(b)**).get());

Exercise:

|  |  |
| --- | --- |
| distinct | * 1. Find distinct employees based on their profession   2. Distinct String   3. Distinct numbers |
| sorted | 1. Natural sort 2. Custom sort 3. Sort based on some specific field of employee |
| limit | Limit only 5 users |
| skip | Skip from 5 to 10 |
| reduce |  |
| min | Minimum number  Minimum salaried employee |
| max | Maximum number  Maximum salaried employee |
| count | Number of items: check whether its getting duplicate or not |
| anymatch |  |
| allmatch |  |
| noneMatch |  |
| findFirst |  |
| findAny |  |

**Realtime scenarios**

1. **How many male and female employees are there in the organization?**
2. **Print the name of all departments in the organization?**
3. **What is the average age of male and female employees?**
4. **Get the details of highest paid employee in the organization?**
5. **Get the names of all employees who have joined after 2015?**
6. **Count the number of employees in each department?**
7. **What is the average salary of each department?**
8. **Get the details of youngest male employee in the product development department?**
9. **Who has the most working experience in the organization?**
10. **How many male and female employees are there in the sales and marketing team?**
11. **What is the average salary of male and female employees?**
12. **List down the names of all employees in each department?**
13. **What is the average salary and total salary of the whole organization?**
14. **Separate the employees who are younger or equal to 25 years from those employees who are older than 25 years.**
15. **Who is the oldest employee in the organization?**

**What is his age and which department he belongs to?**

Sort hashmap: <https://javaconceptoftheday.com/java-8-sort-hashmap-by-keys/>

Collectors: <https://javaconceptoftheday.com/java-8-collectors-tutorial/>

[1. What is a Stream? Stream vs Collection?](https://howtodoinjava.com/java8/java-streams-by-examples/#1-what-is-a-stream-stream-vs-collection)

[2. Creating Streams](https://howtodoinjava.com/java8/java-streams-by-examples/#2-creating-streams)

[2.1. Stream.of()](https://howtodoinjava.com/java8/java-streams-by-examples/#21-streamof)

[2.2. Stream.of(array)](https://howtodoinjava.com/java8/java-streams-by-examples/#22-streamofarray)

[2.3. List.stream()](https://howtodoinjava.com/java8/java-streams-by-examples/#23-liststream)

[2.4. Stream.generate() or Stream.iterate()](https://howtodoinjava.com/java8/java-streams-by-examples/#24-streamgenerate-or-streamiterate)

[2.5. Stream of String chars or tokens](https://howtodoinjava.com/java8/java-streams-by-examples/#25-stream-of-string-chars-or-tokens)

[3. Stream Collectors](https://howtodoinjava.com/java8/java-streams-by-examples/#3-stream-collectors)

[3.1. Collect Stream elements to a List](https://howtodoinjava.com/java8/java-streams-by-examples/#31-collect-stream-elements-to-a-list)

* 1. [Collect Stream elements to an Array](https://howtodoinjava.com/java8/java-streams-by-examples/#32-collect-stream-elements-to-an-array)
  2. Collect stream to a set

[4. Stream Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#4-stream-operations)

[4.1. Intermediate Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#41-intermediate-operations)

[4.2. Terminal operations](https://howtodoinjava.com/java8/java-streams-by-examples/#42-terminal-operations)

[5. Short-circuit Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#5-short-circuit-operations)

[5.1. Stream.anyMatch()](https://howtodoinjava.com/java8/java-streams-by-examples/#51-streamanymatch)

[5.2. Stream.findFirst()](https://howtodoinjava.com/java8/java-streams-by-examples/#52-streamfindfirst)

[6. Parallel Streams](https://howtodoinjava.com/java8/java-streams-by-examples/#6-parallel-streams)

[7. Stream Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#7-stream-operations)

[7.1 Creating Streams](https://howtodoinjava.com/java8/java-streams-by-examples/#71-creating-streams)

[7.2 Intermediate Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#72-intermediate-operations)

[7.3. Terminal Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#73-terminal-operations)

[8. Java Stream Examples](https://howtodoinjava.com/java8/java-streams-by-examples/#8-java-stream-examples)

Reference links

<https://dzone.com/articles/java-8-method-references>

<https://howtodoinjava.com/java8/java-streams-by-examples/#3-stream-collectors>