Features

1. Lambda Expression
2. Functional Interface
3. Default and static methods
4. Predicates
5. Functions
6. Consumer
7. Supplier
8. Method Reference

Lambda expression

What is Lambda expression?

It is anonymous function. A function which don’t have name, access modifier, return type is called anonymous function. Lambda expression is an anonymous function. The type of the lambda expression is a functional interface.

Benefits of lambda expression

1. We can enable Functional programming.
2. To write more readable, maintainable and concise code, we can reduce length of the code so that readability will be improved.
3. We can reduce complexity of anonymous inner class until some point.
4. We can handle functions/procedures like variables.
5. We can pass functions/procedures as an argument.
6. Easier to use updated APIs and libs.
7. Enable support for parallel processing.

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

Functional Interface

What is Functional Interface?

If an interface contains only one abstract method, we can say it as functional interface.

Runnable -> run(), callable -> call are examples for functional interface. Also functional interface have any number of default and static methods. The restriction will only apply for abstract methods only.

Use of @FunctionalInterface annotation

To indicate the interface as functional interface. It is not a mandatory one, but it is a additional option define the functional interface.

Advantage of *Functional* interface annotation

The *functional* interface annotation is an optional one.

When we add another abstract method inside the functional interface the error will thrown at the implementation level(Lambda Expression) If we have @FunctionalInterface annotation compiler will notify at interface level.

It helps to avoid the impact of future modification .

Functional Interface Inheritance

If the parent interface is a functional interface and it extends to child functional interface, and child interface doesn’t have any abstract methods then both parent and child are proper functional interfaces. Since both interface have single abstract methods(We declare abstract method at parent level and same method will available for extended interface).

If the parent interface is a functional interface and it extends to child functional interface, and child interface has one abstract method with same as parent then both parent and child are proper functional interfaces. Since both interface abstract methods(We declare abstract method at parent level and same method override for extended interface).

If the parent interface is a functional interface and it extends to child functional interface, and child interface has one abstract method with different name then child will not be the functional interface. Since it has two abstract methods (We declare abstract method at parent level and same method will available for extended interface also child interface have multiple abstract methods so it won’t be functional interface).

If the parent interface is a functional interface and it extends to child interface(Not functional interface), then child interface can have any number abstract, default and static methods.

If the interface is created with 0 abstract methods, then the compiler error will throw.

Why functional interface should contain only one abstract method?

The purpose of the functional interface is to invoke lambda expression. If there is more than one abstract method then the compiler cannot identify, which method can invoke lambda expression. Then the compiler will throw, “Incompatible type : Interface is not a functional interface” error.

In general, the abstract keyword is not used to define the abstract method inside the interface. Why?

In *the* interface when we define abstract method *implicitly* it considered as Abstract. We don't need to specify *the* abstract keyword.

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

Lambda Expression with Collections

Some of the collection interfaces List, Set, Map.

List

1. Maintains the elements of order in which the order used to insert.
2. List allows duplicate objects.

Set

1. The order is not maintained. Whenever accessing the elements any order can come.
2. Set doesn’t allows the duplicate objects.

Map

1. Map is a key value pair collection .
2. It allows duplicate values but unique keys.
3. If we try to add duplicate key the previous value will be overwritten.

Comparator Interface

1. The lambda expression is used to sort the collection objects. By using Comparator interface, we can achieve it.
2. The Comparator interface has only one abstract method so it is functional interface.
3. Comparator interface helps to define the customized sorting.

Compare method in the Comparator interface

1. It returns negative value if and only if obj1 has to come before obj2.
2. It returns positive value if and only if obj1 has to come after obj2.
3. It returns 0 value if obj1 and obj2 are equal.

Sorting collection user Collections Sort method.

1. In general, the Collections.sort() method receives any of the collection as param. i.e Collections.sort(List l).
2. At that point of time, the collection will sort the element with natural sorting.(i.e If elements are number then sort them with numeric order otherwise alphabetic order).
3. If we passing Comparator as an another argument, the it will sort the objects with the order mentioned in the comparator interface.

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

Anonymous inner class vs Lambda Expression

What is anonymous inner class?

A inner class without any name is caller anonymous inner class. There may be a possibilities to replace the anonymous class with lambda expressions. Lambda expression maintain close relationship with Anonymous inner class.

Q. Can every anonymous inner class replace with lambda expressions?

No, All the anonymous class cannot replace with lambda expressions.

1. An anonymous inner class can extend the concrete class whereas the lambda expression cannot the extent the class.
2. An anonymous inner class can extend the abstract class whereas the lambda expression cannot the extent the class.
3. An anonymous inner class can implement an interface that have multiple abstract methods whereas the lambda expression cannot implement the same.
4. An anonymous inner class can implement an interface that have only one abstract method the lambda expression also implement the same. In this scenario only the anonymous inner class replace with lambda expression.

|  |  |
| --- | --- |
| Anonymous inner class | Lambda expressions |
| It is a class without a name | It is a function without a name |
| An anonymous inner class can extend the concrete and abstract classes | The lambda expression cannot extend the concrete or abstract class. |
| It can implement the interface which contain any number of abstract methods | It can implement the interface which contain only one abstract method. |
| We can create instance variable inside the anonymous inner class | We can create only local variable inside the lambda expressions |
| It can be instantiated | It cannot be instantiated |
| The “this” keyword always refers the current inner class object not outer class object | The “this” keyword always refers the outer class object |
| It is best choice if we want to handle multiple methods | It is best for functional interface |
| At the time of compilation a separate class will be generated | Separate class file won’t create |
| Memory will allocate whenever creating the object | It resides in permanent memory of JVM. |

\*Inside the lambda expression the referenced global variables or implicitly final if it is declared or not.\*

Default and static methods inside the interface

Default methods

1. The concrete methods which allow to declare inside the interface is known as default methods.
2. We can declare default methods using “default” keyword.
3. The default method will be available inside the implementation class by default.
4. Every variable present in the interface is always public static final.
5. If the implementation class want to override the default method it is also possible. In this case, the implementation class method will be refer while calling with object.
6. When implementing multiple interfaces and both have same methods then compiler will throw error.To avoid the compiler error we have to override the method inside the implementation class.

Static methods inside the interface.

1. To define general utility methods, we can use the static methods inside the interface.
2. The static methods inside the interface will not available inside the implementation class.
3. We can’t access these static methods using implementation class objects.
4. Those static methods can call through interface name only (Not by interface objects).
5. Since interface can have static methods, it is also executed without implementation.

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

Predicates

1. A functional interface used to conditional check and returns Boolean values is called predicate.
2. It contains only one abstract method and contains some default methods.
3. The abstract method is “test(T)”.
4. Some default methods are negate(), and(), or().
5. Negate used to apply NOT in the condition.
6. And used to combine two predicates to check both conditions are satisfied or not.
7. Or used to combine two predicates to check any one of the condition is satisfied or not.

Predicate Interface

test() – Abstract method

negate() – Default method

and() – Default method

or() – Default method

Functions

1. A functional interface returns any values is called function.
2. It contains only one abstract method.
3. The abstract method is “apply(T)”.
4. Also the function takes two parameters one is input and another one is return type
5. The function interface contains default(andThen(), compose()) and static(Identity()) methods

|  |  |
| --- | --- |
| Predicate | Functions |
| To implement conditional check | To perform certain operation and to return some result |
| Predicate can take only one parameter input type | Function can take two parameter one is input and another one is return |
| It has only one abstract method “Test” | It has only one abstract method “apply” |
| Public boolean test(T) | Public R apply(T) |
| Returns Boolean value only | It will return any values |

Function Interface

Apply() – Abstract method

andThen() – Default method

compose() – Default method

Identity() – Static method

Function chaining

To combine the two functions the function chaining is used. It can achieve by using andThen and compose functions

Let have functions f1 and f2, if we want to combine these functions

If the f1 should run first and the f2 should run on the result of f2 then we can go with andThen ()

If the f2 should run first and the f1 should run on the result of f2 then we can go with compose ()

Consumer

1. A functional interface returns no values is called consumer.
2. It contains only one abstract method.
3. The abstract method is “accept(T)”.
4. Also the function takes only one parameter.
5. The function interface contains default method andThen().

Consumer Interface

accept() – Abstract method

andThen() – Default method

Consumer chaining

To combine the two consumers the consumer chaining is used. It can achieve by using andThen method

Let have consumers c1 and c2, if we want to combine these consumers we can use c1.andThen(c2)

The c1 will run first and the c2 will run.

Supplier

1. A functional interface returns no values is called supplier.
2. It contains only one abstract method.
3. The abstract method is “get()”.
4. Also the supplier take no parameter.
5. The function interface contains no default methods.

Supplier Interface

get() – Abstract method

Primitive type functional interface

While using functional interface with wrapper object then the autoboxing and auto unboxing will perform by compiler internally. It may affect the performance. To avoid this kind of issue the primitive functional interface is used. This primitive types only apply on the abstract methods. The default and static methods are remains same.

Example – IntPredicate, IntToLongFunction

Unary Operator

If the input and return parameters are same in the function then we can use the unary operator.

Example – IntUnaryOperator => public int applyAsInt(int a)

Binary Operator

If the input and return parameters are same in the Bifunction then we can use the binary operator.

Example – IntBinaryOperator => public int applyAsInt(int a)

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

Method Reference

1. Instead of implementing the interface method, refer the current class method’s implementation using double colon operator (::) is called method reference.
2. Method reference only applicable for the methods which same parameters.
3. The main advantage of method reference is code reusability.
4. Method reference is alternate for lambda expressions
5. We can refer the constructor of the class using :: this is known as constructor reference.

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

Streams

1. To process objects from the collection we can use stream.
2. The stream method is present inside the collection interface as default method.
3. Stream is a interface present in the java.util.stream package.

To configure streams we can use two terminologies

1. Configuration
2. Processing

The Configuration contains two methods

1. Filtering – If we want to filter elements based on some condition in the collection object we can use filter. We can configure filter using filter method of stream interface.

Public Stream filter(Predicate<t> t)

1. Mapping – If we want to create a separate new object for every object present in the collection based on some functions then we can go for mapping. We can implement mapping by using map() method of stream.

Public Stream map(Function<T,R> t)

Stream Processing by collect method

The method collect, collects the elements from the stream and adding to the specified collection.

Stream Processing by count method

The method count, returns the number of elements present in the stream.

Public long count()

Stream Processing by Sorted method

The method sorted, sort the elements present in the stream. We can sort either natural order or customized order.

Sorted() => Natural Sorting

Sorted(Comparator C) => Customized sorting

Stream Processing by forEach method

This method won’t return anything. It will take lambda expression as argument and apply that lambda expression for each element present in the stream.

Stream Processing by toArray() method

To copy elements present in the stream to specified array

Stream Processing by stream() method

We can apply stream for group of values & for arrays