Features

1. Lambda Expression
2. Functional Interface

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\*