Before learning Lambda expression, we should know functional interface.

1. **What is this functional interface?**

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default and static methods but can contain only one abstract method. It can also declare methods of object class (like equals,hashCode,clone..etc).

Any interface with a **SAM** (Single Abstract Method) is a functional interface, and its implementation may be treated as lambda expressions.

A functional interface can extend another interface only when it does not have any abstract method. It throws compile time error.

It's recommended that all functional interfaces have an informative @FunctionalInterface annotation. This clearly communicates the purpose of the interface, and also allows a compiler to generate an error if the annotated interface does not satisfy the conditions.

Example of Java Functional interfaces

@FunctionalInterface  
public interface Runnable {

public abstract void run();  
}

@FunctionalInterface

public interface Comparable<T> {

public int compareTo(T o);

}

@FunctionalInterface  
public interface Comparator<T> {

int compare(T o1, T o2);

}

@FunctionalInterface  
public interface Callable<V> {

call() throws Exception;

….  
}

Our own functional interface

@FunctionalInterface

public interface MyFunctionalInterface{

    public void m1();

    public static void m2(){

        System.out.println("Example0001FunctionalInterface.m2()");

    }

    public static void m3(){

        System.out.println("Example0001FunctionalInterface.m3()");

    }

    public default void m4(){

        System.out.println("Example0001FunctionalInterface.m4()");

    }

    public default void m5(){

        System.out.println("Example0001FunctionalInterface.m5()");

    }

}

1. **Java 8 Anonymous Function Example**

Anonymous function means a method with no name and no modifier.

An Anonymous is a nameless function.

Java is an object-Oriented Programming Language, and as such concepts like Encapsulation and Abstraction sits at the heart of it. However, since Java is a bit-too-much Object Oriented at its genome structure, that often times than not, it adds a lot of verbosity and boiler plate to its language constructs.

Using Interface implementation

public interface Calculator {

   int add(int *a*, int *b*);

}

public class CalculatorImpl implements Calculator{

     public int add (int *a*, int *b*){

         return a+b;

     }

    public static void main(String[] *args*){

        Calculator calculator = new CalculatorImpl();

        System.out.println(calculator.add(3,4));

    }

}

Using anonymous function.

We no need to implement the interface. We can write implementation using anonymous function

public class AnonymousTest {

    public static void main(String[] *args*){

    Calculator calculator = new Calculator (){

                                  public int add (int *a*, int *b*){

                                     return a+b;

                                  }

                               };

    System.out.println(calculator.add(3,4));

    }

}

**Anonymous classes are inner classes with no name.** Since they have no name, we can't use them in order to create instances of anonymous classes. As a result, we have to declare and instantiate anonymous classes in a single expression at the point of use.

We may either extend an existing class or implement an interface.

**Extend a Class**

When we instantiate an anonymous class from an existent one, we use the following syntax:

[](https://www.baeldung.com/wp-content/uploads/2019/04/AnonymousClass-InstantiateFromClass.png)

In the parentheses, we specify the parameters that are required by the constructor of the class that we are extending:

**new** Book("Design Patterns") {

@Override

**public** String **description**() {

**return** "Famous GoF book.";

}

}

Naturally, if the parent class constructor accepts no arguments, we should leave the parentheses empty.

**2.2. Implement an Interface**

We may instantiate an anonymous class from an interface as well:

[](https://www.baeldung.com/wp-content/uploads/2019/04/AnonymousClass-InstantiateFromInterface.png)

Obviously, Java's interfaces have no constructors, so the parentheses always remain empty. This is the only way we should do it to implement the interface's methods:

**new** Runnable() {

@Override

**public** **void** **run**() {

...

}

}

Once we have instantiated an anonymous class, we can assign that instance to a variable in order to be able to reference it somewhere later.

We can do this using the standard syntax for Java expressions:

Runnable action = **new** Runnable() {

@Override

**public** **void** **run**() {

...

}

};

As we already mentioned, **an anonymous class declaration is an expression, hence it must be a part of a statement**. This explains why we have put a semicolon at the end of the statement.

Obviously, we can avoid assigning the instance to a variable if we create that instance inline:

List<Runnable> actions = **new** ArrayList<Runnable>();

actions.add(**new** Runnable() {

@Override

**public** **void** **run**() {

...

}

});

We should use this syntax with great care as it might easily suffer the code readability especially when the implementation of the run() method takes a lot of space.

Anonymous classes cannot have any static members except for those that are constant.

For example, this won't compile:

**new** Runnable() {

**static** **final** **int** x = 0;

**static** **int** y = 0; // compilation error!

@Override

**public** **void** **run**() {...}

};

Instead, we'll get the following error:

The field y cannot be declared **static** in a non-**static** inner type, unless initialized with a constant expression

1. **Lambda expression**

Lambda expression were introduced in Java 1.8. This is one of the best features of Java (of course lambda expression is already part of many programming languages, but in Java they are introduced in Java 1.8).

We can write java code in functional style using Lambda expression.

Anonymous function syntax:

|  |  |  |
| --- | --- | --- |
| Parameter | Expression | Body |
| () | -> | { } |
| (I,j) | -> | {return i+j}  Or {i+j} |

It’s an expression through which we can represent an anonymous function.

Examples:

    System.out.println(new Object() {

        @Override public String toString() {

            return "Hello world!";

        }

    });

Comparator<String> c = new Comparator<String>() {

              int compare(String *s*, String *s2*) { ... }

            };

        }

 Runnable action = new Runnable() {

    @Override

    public void run() {

        System.out.println("run method");

    }

    };

Normal method

 public void m1(){

        System.out.println("hello");

    }

Anonymous function (same method above can be represented as below)

m1()->System.out.println("hello");

How to call anonymous function?

We don’t have any name for our Anonymous function, then how can we call it? That is where functional interfaces come into picture. We can write lambda expressions for the abstract method in functional interface

Traditional way

public class MyFunctionalInterfaceTest implements MyFunctionalInterface{

    public void m1(){

        System.out.println(" method implementation");

    }

    public static void main(String[] *args*){

        MyFunctionalInterfaceTest  test = new MyFunctionalInterfaceTest();

        test.m1();

    }

}

Lambda way

public class MyFunctionalInterfaceTest {

    public static void main(String[] *args*){

        MyFunctionalInterface  test = ()->{System.out.println(" method implementation");};

        test.m1();

    }

}

Its easy and simple in lambda way (just on line)

Runnable task2 = () -> { System.out.println("Task #2 is running"); };

Comparator<Product> nameComparator = (Product o1, Product o2)-> o1.getName().compareTo(o2.getName());

    List<Product>  productList = new ArrayList<>();

    productList.add(new Product());

    ....

    productList.sort((Product o1, Product o2)->o1.getName().compareTo(o2.getName()));