Functional Interface

In Java, an interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. They cannot contain instance variables (fields). An interface is defined using the `interface` keyword.

A **Functional Interface**, also known as a **Single Abstract Method Interface (SAM Interface)**, is an interface that contains exactly one abstract method. The purpose of a functional interface is to be used with **lambda expressions**, which are a feature introduced in Java 8.

Why Functional Interfaces?

Functional interfaces and lambda expressions are key to functional programming in Java. They let you treat actions as objects, meaning you can pass actions (like methods) as parameters, store them in data structures, and so on. This can lead to more flexible and reusable code.

How to Define a Functional Interface

You can define a functional interface like this:

```
@FunctionalInterface
public interface Printable {
    void print();
}
```

In this example, `Printable` is a functional interface because it has exactly one abstract method: `print()`. The `@FunctionalInterface` annotation is optional but recommended, as it makes the compiler check that the interface really does have exactly one abstract method, and will throw an error if not.

Anonymous Class Definition

If I have a Functional Interface `Printable` mentioned earlier, we can implement the interface from a class called `EpsonPrinter` and then do the normal job of implementing the interface method.

```
public class EpsonPrinter implements Printable{

    public EpsonPrinter() {}

    @Override
```

```
    public void print() {
        System.out.println("Ami Valo Achi");
    }
}
```

Now in main, we can call the object of `EpsonPrinter` and call the function `print()`

```
public static void main(String[] args) {
    Printable p = new EpsonPrinter();
    p.print();
}
```

The output is

```
Ami valo achi
```

Now if we do not need EpsonPrinter and need the implementation of the method that is just hard coded to print “Ami valo achi” and also that is for only once. We can make that implementation inside of the main function itself without giving it a name. This is an anonymous class.

```
public class Main {
    public static void main(String[] args) {
        Printable p = new Printable() {
            public void print() {
                System.out.println("Ami valo achi");
            }
        };

        p.print();
    }
}
```

This code is a simple Java program that prints out a message. Here’s what it does:

1. It defines a class named `Main`. This is the main class of the program.
2. Inside the `Main` class, it defines a method named `main`. This is the entry point of the program. When you run the program, this is the first method that gets called.
3. Inside the `main` method, it creates an instance of an anonymous class that implements the `Printable` interface. This anonymous class overrides the `print` method of the

`Printable` interface to print out the message “Ami valo achi”.

4. It then calls the `print` method on the instance of the anonymous class. This causes the message “Ami valo achi” to be printed out.

So, when you run this program, it will print out the message “Ami valo achi”. The `Printable` interface and the anonymous class are just a way to organize the code and make it more modular and reusable. The important thing is the `print` method, which contains the code that gets executed when the program runs. In this case, that code is `System.out.println("Ami valo achi");`, which prints out the message.

Lambda Expression

Now what Java says is,

“I know what you are going to put after you write `Printable p =` So there is no need to write `new Printable()` in your code. I can assume that.”

Java also says that,

“ Since you are referencing a functional interface I know what is the access modifier, what is the return type, what is the parameter input type and what is the name of the method. So you can remove that and I will assume those also.”

After this, our code will look like this,

```
public class Main {
    public static void main(String[] args) {
        Printable p = () → {
                System.out.println("Ami valo achi");
            };
        p.print();
    }
}
```

Now the → you see that indicates Java to look for Lambda Expression. As the function has only one line we can get rid of the curly braces and take everything in one line,

```
public static void main(String[] args) {
    Printable p = () → System.out.println("Ami valo achi");
    p.print();
}
```

And the output will be

```
Ami valo achi
```

The process of shrinking obvious code from more than 15 lines to just 1 line is the magic of lambda expression.

Lambda Expression With Parameters

If our function has some parameters. Suppose our function is ‘add’ and its part of the adder functional interface.

```
@FunctionalInterface
public interface Adder {
    int add(int a, int b);
}
```

Now in the main function where we want to use our lambda expression, we write

```
public static void main(String[] args) {
    Adder ad = (a, b) -> a + b;
    int x = ad.add(10, 20);
    System.out.println(x);
}
```

Here we do not need the parameter return type because Java can specify that from the interface. We also do not need to state the word `return` because we are doing $a + b$ that means we are gonna return that. Java assumes it.

So the Output of this program is

```
30
```

The last view on lambda expression

Lambda expressions, introduced in Java 8, are a significant addition to the Java programming language. They provide a clear and concise way to represent a method interface using an expression. Lambda expressions are similar to methods, but they do not need a name and can be implemented right in the body of a method.

The primary purpose of lambda expressions is to define the inline implementation of a functional interface, i.e., an interface with a single method only. In the context of collection libraries, lambda expressions help to iterate, filter, and extract data.

Lambda expressions have the following syntax:

```
(argument-list) -> {body}
```

The argument list can be empty or non-empty as well. The body part can contain a single statement or a statement block. The body part needs to have curly braces `{}` in case of a multi-line statement and no curly braces are required for a single line statement.

Lambda expressions can be used as method arguments or to define the result of a higher-order function. They can also be used to condense many lines of code into a single line, making the code more readable and efficient.

In conclusion, lambda expressions in Java help to write more efficient and readable code by reducing the verbosity of traditional anonymous classes. They have become a fundamental part of Java, particularly since the introduction of stream operations in Java 8.