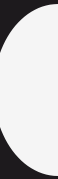


FUNCTIONAL PROGRAMMING



Imperative Vs Declarative Style

- Imperative style in which we tell Java “every step of what you want it to do and then you watch it to exercise those steps”.
 - Feels bit low level.
 - Lack of intelligence.
- Declarative style in which you tell “What you want” rather how to do it. Declare your desired results, but not step by step.

Imperative way Example:

Find if **Chicago** is in the collection of given **cities**.

```
boolean found = false;
for(String city: cities){
    if (city.equals("Chicago")){
        found = true;
        break;
    }
}
System.out.print("Found chicago?: " + found);
```

This imperative version is noisy and low level.

First initialize a boolean flag and then walk through each element in the collection.

If we found the city we're looking for, then we set the flag and break out of the loop.

Finally we print out the result of our finding.

A Better way:

As observant Java programmers, the minute we set our eyes on this code we'd quickly turn it into something more concise and easier to read, like this:

```
System.out.println("Found chicago?: " +  
    cities.contains("Chicago"));
```

This is one very simple example of declarative style—the **contains()** method helped us get directly to our business.

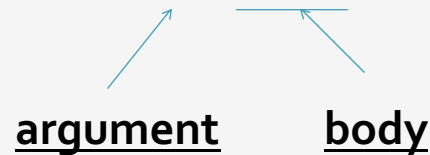
- Introduced in Java 8.
 - Greek letter.
 - Lambda calculus.
 - Main feature in Java 8.
 - OOP also got Functional touch.
-



History: Lambda Calculus

- Mathematical notation for functions.
- Introduced by **Alonzo Church** in 1930s.
- Lambda-calculus functions -->
anonymous

$\lambda x. x * x$



- In 1950s, **John McCarthy** invented **LISP** at MIT.
- McCarthy is also the founder of AI field.
- LISP: Models mathematical problems.
- Heavily influenced by lambda-calculus.

(lambda(arg)(*arg arg))

- Functional programming has roots in lambda-calculus.
-

What is Lambda?

- Anonymous function.
- Compact way to define functions.
- Useful for *passing around functionality*.
- LISP, Scala, C#, Ruby, C++, Python



Lambda Syntax

$(Type\ param1, Type\ param2, \dots) \rightarrow \{$

`// statement 1`

`// statement 2`

`....`

`return;`

`}`


Lambda

- Lambda can be assigned to a variable whose type is of **functional interface**.

functional interface variable $\leftarrow \lambda$

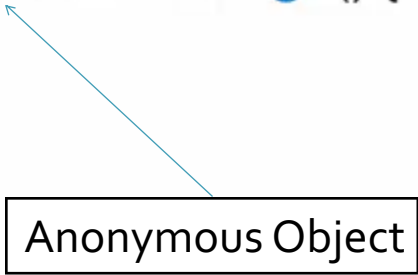
- Functional interface possess a single **abstract method**, (Single Abstract Method interface)

Method parameter



Example

```
Set<String> set = new TreeSet<String>(new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```



TreeSet<String>((String s1, String s2) → {return s1.length() - s2.length();})

Further simplification

→ (s1, s2) → {return s1.length() - s2.length();}

More simplification
If the body has only
One statement

→ (s1, s2) → s1.length() - s2.length()

- Comparator Interface is of type Functional Interface if not then we get compilation error.
- lambda expression can't be assigned to a method parameter or variable whose type is not functional interface.

Anonymous Classes vs Lambda

- Before java 8 anonymous classes played the role of lambdas.

Has associated object +
verbose

Instantiated on every use
(unless declared as **Singleton**
by using static or final)

Target type (class/ interface) can
have multiple methods

No associated object +
Compact representation

Memory allocated only once for method

Works only with functional interface



Demo

Check the
uploaded sample
code on
blackboard for
Lambdas demo

Simplifications

//Lambdas

```
Arrays.sort(recommendedItems.items, (Bookmark o1, Bookmark o2) -> {  
    return new Integer(o1.getTitle.length()).compareTo(new Integer(o2.getTitle.length()));  
});
```

```
System.out.println("\nSorted by length (using lambdas) ...");  
    iterator = recommendedItems.iterator();  
    while (iterator.hasNext()) {  
        System.out.println(iterator.next().getTitle());  
    }  
}
```

```
1. Arrays.sort(recommendedItems.items, (o1, o2) -> {  
    return new Integer(o1.getTitle.length()).compareTo(new  
    Integer(o2.getTitle.length()));  
});
```

```
2. Arrays.sort(recommendedItems.items, (o1, o2) ->  
    new Integer(o1.getTitle.length()).compareTo(new  
    Integer(o2.getTitle.length())));
```

Why code in Functional

Style?
Is it worth picking up the new style?

Should we expect a marginal improvement, or is life altering?

- Writing Java code is not that hard; the syntax is simple.
- What really gets us is the effort required to code and maintain the typical enterprise applications we use Java to develop.
- We must ensure the database connection is closed properly and on time.
- One must not hold on a transaction longer than they needed it.
- Exceptions are handled well.
- Securing and releasing the locks properly.
-



How to introduce Lambda expression in Java using

1. create your own functional interface.
2. Use the pre-defined functional interfaces in Java



Functional interface

1. Functional interface is a Java interface with *single abstract method*.
 2. Use **@FunctionalInterface** annotation to explicitly mark it.
-

Lets Break it Down a Syntax: bit

(parameters) -> {expression body}

Methods in Java:

1. Name
2. Parameter list
3. Body
4. Return Type

Lambda Expression:

1. **No** name – function is anonymous
 2. Parameter list
 3. Body – main part of the function
 4. **No** return type – java 8 compiler is able to infer the return type by checking the code.
-

<code>() -> {};</code>	No parameters, Empty block
<code>() -> {System.out.println("Hello");};</code>	No parameters, Block statement
<code>() -> 24</code>	No parameters, return integer 24
<code>(int x, int y) -> x + y;</code>	Two integers, return the sum
<code>(Object x) -> x</code>	Given an object, it returns it
<code>(Person p) -> p.age > 25 && p.salary < 2000</code>	Given reference to Person returns boolean
<code>n -> n % 2 == 0;</code>	Given a number returns a boolean
<code>(String s1, String s2) -> s1.length() + s2.length();</code>	Given two strings return an integer

Let suppose we want to create a program in which we want to return true only if the sum of the given two integers are even.

Therefore we create an interface with only one method that takes two integers and returns a boolean value.

```
boolean evenSum(int x, int y);
```

```
@FunctionalInterface
```

```
public interface Summable{
```

```
/**
```

```
 * Returns true only if the sum of params is even
```

```
 *
```

```
 * @param x the integer operand
```

```
 * @param y the integer operand
```

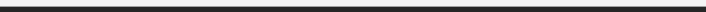
```
 * @return true if the sum of x and y is an even number
```

```
 */
```

```
boolean evenSum(int x, int y);
```

```
}
```

First way to solve the problem



```
public class FirstWay implements Summable{

    /**
     * The implementation of evenSum
     * defined in Summable interface
     * @param x the integer operand
     * @param y the integer operand
     * @return true if the sum of x and y is an even number */

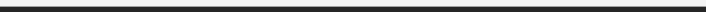
    @Override
    public boolean evenSum(int x, int y) {
        return (x + y) % 2 == 0;
    }

    public static void main(String[] args) {

        //create the obj of type Summable
        Summable obj = new FirstWay();

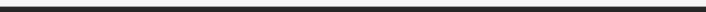
        //invoke method eventSum and print the result
        System.out.println("Is sum even? " + obj.evenSum(1, 2));
    }
}
```

Second way to solve the problem



```
public class SecondWay {  
    public static void main(String[] args) {  
  
        //anonymous class  
        //create the object of type Summable and invoke eventSum on it  
  
        System.out.println("Is sum even? " + new Summable() {  
  
            @Override  
            public boolean evenSum(int x, int y) {  
                return (x + y) % 2 == 0;  
            }  
        }.evenSum(1, 2));  
    }  
}
```

Lambda Expression Way



```
public class ThirdWay {  
  
    public static void main(String[] args) {  
  
        //create an obj of type Summable using a lambda expression:  
        //(x, y) -> { return (x + y) % 2 == 0; };  
  
        Summable obj = (x, y) -> { return (x + y) % 2 == 0; };  
  
        System.out.println("Is sum even? " + obj.evenSum(1, 2));  
    }  
}
```

Lambda with Multiple parameters

```
interface StringConcat {  
    public String sconcat(String a, String  
b);  
}  
  
public class Example {  
    public static void main(String args[]) {  
        // lambda expression with multiple  
arguments  
StringConcat s = (str1, str2) -> str1 + str2;  
  
System.out.println("Result:  
"+s.sconcat("Hello ", "World"));  
    }  
}
```

