Functional Programming

By: Mahboob Ali

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.

- Meaning Computation’s logic is expressed without describing its controls flow, SQL and Regular Expressions are both declarative style examples.

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.

λ

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

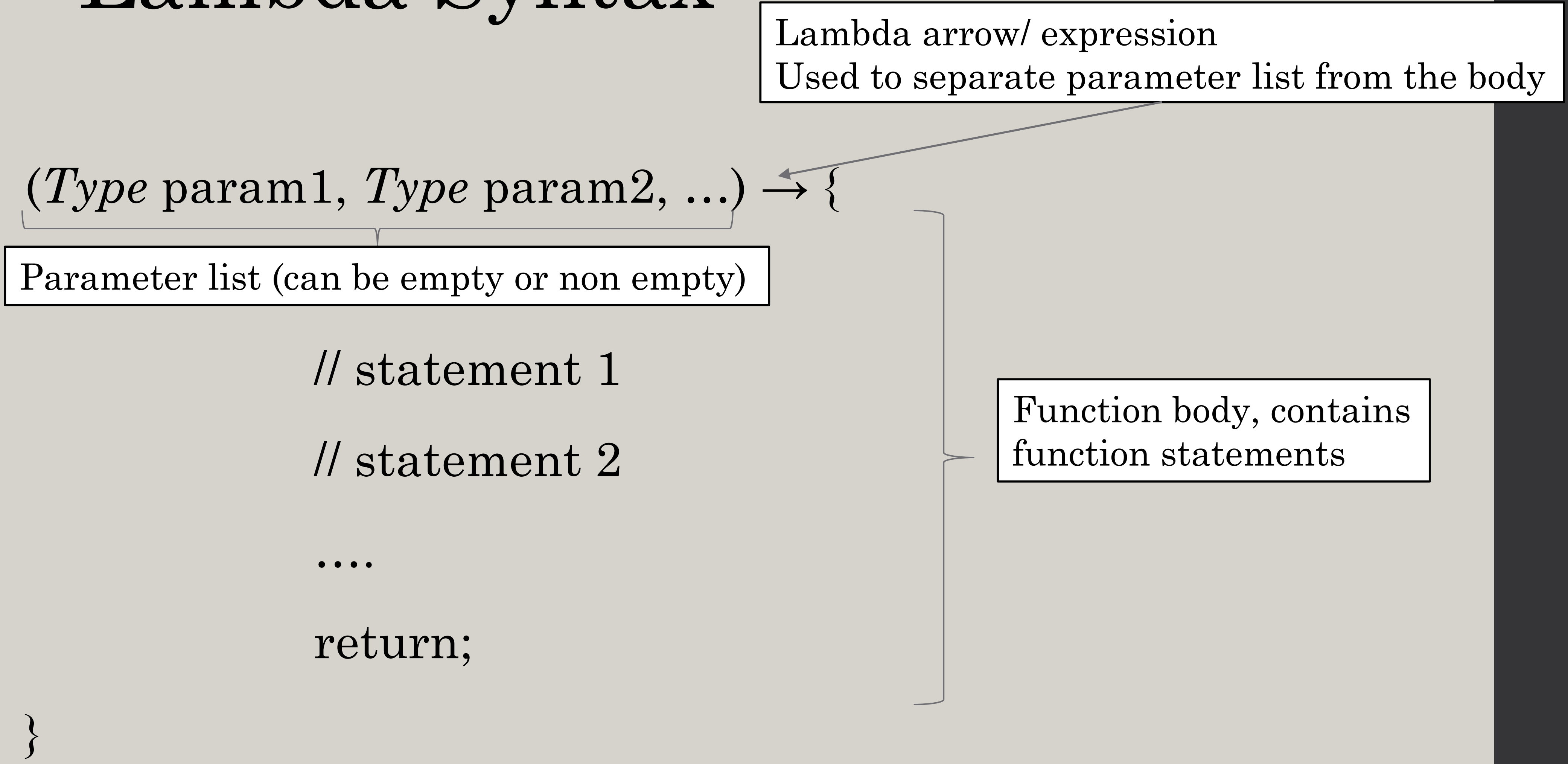
Functional Programming

- Computations in functional programming can be described as *functions* which are evaluated as expressions.
- These kind of functions follow the mathematical flow of functions, like there output depends totally on arguments.
- Doesn’t matter how many times you call them (with same arguments) they will produce the same results.
- Functional programming favors ***immutability*** → **the state can’t change**
- Imperative programming functions (normal java functions) might be associated with state (such as java instance variables).

What is Lambda?

- Anonymous function.
- Compact way to define functions.
- No name
- Doesn’t belong to any class.
- It simplified the development by facilitating the *functional interface*.
- LISP, Scala, C#, Ruby, C++, Python`

Lambda Syntax



```
(Type param1, Type param2, ...) → {

                // statement 1

                // statement 2

                ....

                return;

}
```

Lambda arrow/ expression
Used to separate parameter list from the body

Parameter list (can be empty or non empty)

Function body, contains
function statements

Some characteristics

- Type declaration in lambda expression is optional, java automatically decide the type depending on the parameter list, for example

  `(10,11) ->` *function body*

- If the parameter is only one you can omit the parenthesis as well. Like

  `10 ->` *function body*

- Similarly if there is only statement in the body you can omit the curly braces of the body as well, like

  ```
  (10,11) -> 10 + 11;
  ```

- Return statement is also optional, java automatically return the value if the body has a single expression. If the function body return the value then you need body curly braces as well.

Anonymous Classes Vs Lambda

Before Java 8 anonymous classes played the role of lambdas.

Has associated object + verbose	No associated object + Compact representation
Instantiated on every use (Unless declared as Singleton by using static or final)	Memory allocated only once for a method
Target type (class/ interface) can have multiple methods	Works only with functional Interface

Lets Break it Down a bit

Syntax: (parameters) -> {expression body}

Methods in Java	Lambda Expression
Name	No Name
Parameter List	Parameter List
Body	Body (main part of the function)
Return Type	No return type (java 8 compiler is able to infer the return type by checking the code)

How to introduce Lambda expression in Java

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.

- Lambda can be assigned to a variable who’s type is of **functional interface**.

    functional interface variable ← λ

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

Lambda Expression with No Parameter

- Lets take an example in which we want to create a Functional Interface which has a method Hello with no parameters.

```
@FunctionalInterface
interface NoParameterInterface{
     public String Hello(); //method with no parameter and
                            //return a string
}
```

- Now we have to create a lambda expression where we can use this method.

```
public class NoParameterClass{
    public static void main(String[] args){
        //lambda expression with return
        NoParameterInterface msg = () -> {
            return "Hello world from lambda";
        };

    System.out.println(msg.Hello());
    }
}
```

Lambda Expression with One Parameter

- Lets take another example in which we want to create a Functional Interface which has a method Square with one parameters.

```
@FunctionalInterface

interface OneParameterInterface{

     public int SquareValue(int value);

}
```

- Now we have to create a lambda expression where we can use this method.

```
public class OneParameterClass{
     public static void main(String[] args){
          //lambda expression with return
          OneParameterInterface square = (num) ->
                                   num * num;
     System.out.println(square.SquareValue(5));
     }
}
```

- Return statement of the lambda is omitted as it has only one statement.

- 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

```java
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));
        }
}
```

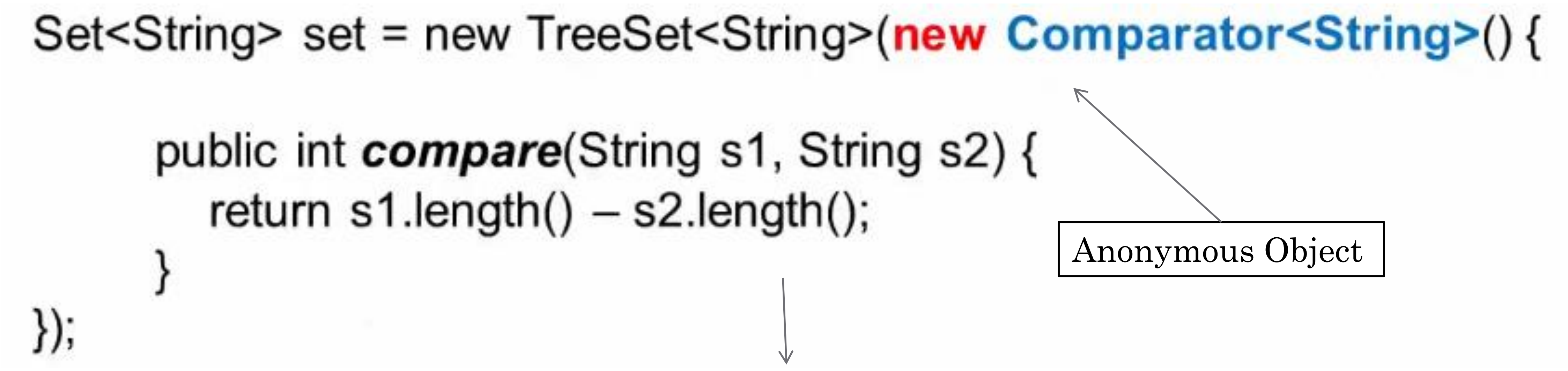
LAMBDA EXPRESSION WAY

```
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"));
        }
}
```

Example



```
Set<String>  set = new TreeSet<String>(new Comparator<String>() {

        public int compare(String s1, String s2) {
            return s1.length() – s2.length();
        }
});
```

Anonymous Object

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 who's type is not functional interface.