

Functional Programming (Lambda Expression)

B.Tech. (IT), Sem-6,
Applied Design Patterns and Application Frameworks (ADPAF)

Dharmsinh Desai University
Prof. H B Prajapati

Last updated: 26 Jan 2021

1

Topics

- Lambda Expression
- Default methods
- Using lambdas in design patterns

2

Lambda Expression

3

Introduction to Functional Programming

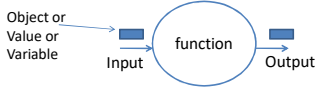
- In Object Oriented Programming
 - Everything is an **object (i.e, noun or data)**
 - E.g., Student `s = new String();` // `s` is a noun
 - Even a method (**action**) does not exist outside an object
 - If we think do define some method, we cannot think without creating a class.
 - An object can be passed to a method or an object can be returned from a method

4

Introduction to Functional Programming

- If we have the following:

```
String msg = "Hello World";
int count = 40;
```


 – We can pass these **values** to any function
 

```

graph LR
    Input[Object or Value or Variable] --> Function((function))
    Function --> Output[ ]
      
```
- But can we consider any function as a value?
 - i.e., can we write, `someVar = someFunction() { ... }`
- And, then we can pass `someVar` (which now actually is a **function**) **as a value** to any function?

5

What is Functional Programming?

- Functional Programming allows us to treat a function (**action**) as a value.
- It allows us to pass an action or behaviour in the same way as we pass a value to a function.
 

```

graph LR
    Input[Action or Behaviour or Function] --> Function((function))
    Function --> Output[ ]
      
```
- The action/behaviour that we pass as an argument is represented as a **lambda** in most of the programming languages.

6

Functional Programming in Java

- Let us understand how **lambda syntax** is derived in Java:
- We define a method in Java as

```
public void myFunction() {
    System.out.println("Hello World");
}
```
- If we can assign this **function** as a **value** to some variable, it would be as follows:

```
funcVar = public void myFunction() {
    System.out.println("Hello World");
}
```

7

Functional Programming in Java

- The **public** keyword makes sense only for a method defined inside a **class**
- We have a standalone method, so it can be dropped

```
funcVar = public void myFunction() {
    System.out.println("Hello World");
}
```
- Even, **return type** (i.e., **void**) is not required to be specified, as the **compiler** can **infer** it from the function **body**.

```
funcVar = public void myFunction() {
    System.out.println("Hello World");
}
```
- Here there is **no return statement**, it means return type is **void**.

8

Functional Programming in Java

- The **name of function** (myFunction) is not used, as we can refer the function using funcVar, so myFunction word can also be dropped

```
funcVar = public void myFunction() {
    System.out.println("Hello World");
}
```
- Thus, we are left with the following:

```
funcVar = () {
    System.out.println("Hello World");
}
```

9

Functional Programming in Java

- We are left with the following:

```
funcVar = () {
    System.out.println("Hello World");
}
```
- Lambda expression is made by adding an arrow, as shown in the following way:

```
funcVar = () -> { System.out.println("Hello World"); }
```
- Thus, funcVar (on left) is now holding a function (on right) as a value.
- Now, we will be able to pass function/behaviour (funcVar) to any function.

10

Functional Programming in Java

- ```
String msg = "Hello World";
int count = 40;
```
- Since, Java is a typed language, we specified **String** as a data type for value **"Hello World"** and **int** as a data type for value **40**
  - Same way in the following expression:  

```
funcVar = () -> { System.out.println("Hello World"); }
```
  - There should be some data type for funcVar  
**fun-data-type** funcVar = () -> {  
     System.out.println("Hello World");  
}
  - What should be the data type for fun-data-type ?  
 – We will discuss it after discussing the motivations for functional programming in Java

11

## Need for new API (Stream)

- Java 8 (support of functional programming) was introduced in 2014
- Motivation for new API (Stream) was
  - BigData: demanding massive data processing
  - Multit-core Processors as norms
- Main challenge was how to **add some methods** in **existing API**, e.g., in Collection, **without breaking existing** implementation **classes**.
- Much earlier Java had realized that **functional programming** should be there is Java, other languages already supported, e.g., C (using function pointer).
- Java provided a new way of performing computation (using Stream processing-the root cause) and introduced the following:
  - Lambda expression and Method Reference
  - Default methods

12

## Reasons for lambda expression

- There was no way to define a **method without a class**.
  - There was no way to **pass a method as an argument** to a method or **returning a method body** from another method.
  - A new language feature called “**Lambda Expression**” is introduced in Java 8 to overcome these limitations.
  - Lambda enables feature of functional programming in Java language.

13

## What is Lambda Expression?

- `() -> { stmt; }`
- Lambda expression is an **anonymous function**.
  - An anonymous function means a method
    - **Without a declaration**, i.e., without
      - Name declaration
      - Access modifier
      - Return value declaration
  - An anonymous function saves our effort of declaring and writing a separate method to the containing class.
  - Example, we want to write a method that takes no arguments and prints “Hello World” message
 

```
() -> System.out.println("Hello World");
```
  - Lambda allows to pass functionality (function or code) around (as parameter or return value)

14

## Reason for default methods

- Without breaking existing interfaces how to add features?
  - If methods, without body, are added in interfaces, all existing classes (implementing those interfaces) become unusable.
- How to provide a mechanism to add new methods to existing interfaces without breaking backward compatibility?
- Java added default implementation of methods in interfaces.
- Thus, these default methods, provided in root interfaces, become available in the classes automatically.

15

## What is the effect of default methods?

- Interfaces in Java 8
  - Java interfaces are not really interfaces anymore
    - **They (can) provide implementation !!!**
- Interface in Java 8 provide **default implementation** for those methods that
  - Don't need state.
- Default method will not be using any variables of interface and hence a default method works on its input arguments.

16

## Example: Passing a method in old way (Prior Java 8)

- Interface
 

```
public interface WorkerInterface {
 abstract public void doSomework();
}
```
- Using the interface
 

```
public class WorkerInterfaceUser {
 public void execute(WorkerInterface workI){
 workI.doSomework();
 }
}
```

17

## Example: Passing a method in old way (Prior Java 8)

- Test class
 

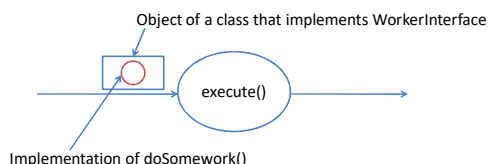
```
public class Test {
 public static void main(String[] args) {
 WorkerInterfaceUser workerIUser=new WorkerInterfaceUser();
 // Now we want to call execute() method of WorkerInterfaceUser
 // We do following in traditional way
 workerIUser.execute(new WorkerInterface() {
 @Override
 public void doSomework() {
 System.out.println("do some work using anonymous class");
 }
 });
 }
}
```

18

### Example: Passing a method in old way (Prior Java 8)

- What are we passing for WorkerInterface?

```
public interface WorkerInterface {
 abstract public void doSomework();
}
```



19

### Example: Passing a method in old way (Prior Java 8)

- Code with anonymous class
 

```
workerUser.execute(new WorkerInterface() {
 @Override
 public void doSomework() {
 System.out.println("do some work using anonymous class");
 }
});
```
- Essentially, the argument to workerUser.execute() method is an anonymous object of an anonymous class that implements WorkerInterface
- Note: In Java, we used anonymous classes in event handling.

20

### Example: Passing a method in old way (Prior Java 8)

Code without anonymous class

- Create a separate implementing class
 

```
class WorkerImpl implements WorkerInterface{
 public void doSomework() {
 System.out.println("do some work using anonymous class");
 }
}
```
- Then call execute method
 

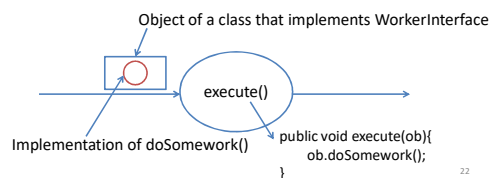
```
workerUser.execute(new WorkerImpl());
```

21

### Example: Passing a method in old way (Prior Java 8)

- Essentially using the following code, what are we doing?
 

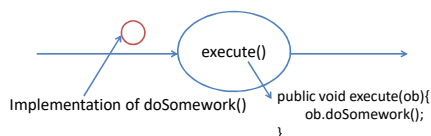
```
workerUser.execute(...);
```
- We are actually **passing** a method **definition of doSomework()** of WorkerInterface to workerUser.execute() method.



22

### Example: Passing a method in old way (Prior Java 8)

- Is there any way to pass only method implementation (not even its signature needed) to workerUser.execute() method?
- The answer is using Lambda expression available in Java 8.
  - But it requires that WorkerInterface must have **only one method declaration**.
  - Such interface is called as **functional interface**.



23

### Example: passing a method body using lambda expression (Java 8)

- Functional interface can have only one non-default (**abstract**) method, but it can have one or more default methods.

```
@FunctionalInterface
public interface WorkerInterface {
 abstract public void doSomework();
}

// User of the functional interface
public class WorkerInterfaceUser {
 public void execute(WorkerInterface workl){
 workl.doSomework();
 }
}
```

24

## Example: passing a method body using lambda expression (Java 8)

- How to write lambda
  - If only one statement we can write  
`() -> stmt;`
  - If more than one statement, we need to write  
`() -> {stmt1; stmt2; ... stmtN};`
- Pass method body (highlighted text) using lambda

```
public class Test {
 public static void main(String[] args) {
 WorkerInterfaceUser workerUser=new WorkerInterfaceUser();
 workerUser.execute(
 () -> {System.out.println("do some work using lambda expression");}
);
 }
}
```

25

## Passing a method body using lambda expression (Java 8)

- Passing a method body using lambda works only if the interface has only one method declared.
- If there were **two declarations of methods**, then **lambda cannot work** due to following reason
  - Using lambda, we pass only one method (implementation). But, if the interface has more than one method, how to pass only one method without implementing the other? (i.e., anonymous inner class must define both methods)
  - If two declared methods have same signature, which one we are referring.

```
@FunctionalInterface
public interface WorkerInterface {
 abstract public void doSomework();
 abstract public void doImportantwork();
}
```

then lambda `() -> {}` refers to which implementation?

26

## Functional interface

- FunctionalInterface must have only **one abstract method**.
- @FunctionalInterface causes the Java 8 compiler to produce an error if the interface has more than one abstract method. It is an **optional annotation**, but is useful to avoid mistakes.

```
7
8
9
10
11
12
13
14
15
16
17
@FunctionalInterface
public interface WorkerInterface {
 abstract public void doSomework();
 abstract public void doImportantwork();
}
```

27

## Example: passing a method body with arguments using lambda expression

- An interface having one method taking two arguments:

```
package mathoperationwithlambda;

@FunctionalInterface
public interface MathOperation{
 int operation(int a, int b);
}
```

28

## Example: passing a method body with arguments using lambda expression

```
package mathoperationwithlambda;

public class Test {
 public static void main(String[] args) {
 MathOperation addition= (int a, int b)-> a+b;
 MathOperation subtraction= (int a, int b)-> a-b;
 MathOperation multiplication= (int a, int b)-> {return a*b;};
 MathOperation division= (int a, int b)-> a/b;

 System.out.println("10 + 25 = "+operate(10, 25, addition));
 System.out.println("10 - 25 = "+operate(10, 25, subtraction));
 System.out.println("10 * 25 = "+operate(10, 25, multiplication));
 System.out.println("10 / 25 = "+operate(10, 25, division));
 }

 static private int operate(int a, int b, MathOperation mathOperation){
 return mathOperation.operation(a, b);
 }
}
```

29

## Running the Example: passing a method body with arguments using lambda expression

```
Notifications Output - MathOperationWithLambda (run)
run:
10 + 25 = 35
10 - 25 = -15
10 * 25 = 250
10 / 25 = 0
```

30

## What is essentially a lambda expression

- Example of lambda expression
  - `MathOperation addition= (int a, int b)-> a+b;`
- We assign **lambda expression** to a **reference of functional interface**.
- **Right hand side** is the **body of the function** whose **signature** is declared in the **functional interface**.
- But, using a lambda which method we are defining?
  - Since functional interface has one method, that method's signature is the signature of the body that you are attaching using lambda
- **Function signature** is in **functional interface** and **function body** is in **lambda expression**.

31

## What is essentially a lambda expression

- We can use any name for the argument.
- ```
@FunctionalInterface
public interface MathOperation{
    int operation(int a, int b);
}
```
- Using lambda


```
MathOperation addition= (int x, int y)-> x+y;
```
 - One more thing about functional interface

Abstract class with one method is not a functional interface.

32

Method References

- Method references allow to use existing method as a value. We use `::` (colon twice) to indicate method reference.
- If a method already exists, we can use it and can assign it in place of lambda expression.
- For example, if we have a lambda expression:


```
(Student s) -> s.getId()
```
- We can straight way right `Student::getId` in place of the above lambda expression
- Thus, a method reference is shorthand for a lambda expression

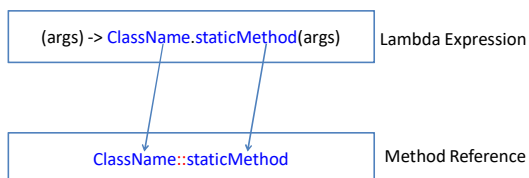
33

Types of Method References

- Three main kinds of method references:
 1. A method reference to a static method
 2. A method reference to an instance method (Instance is coming via parameter)
 3. A method reference to an instance method of an existing object

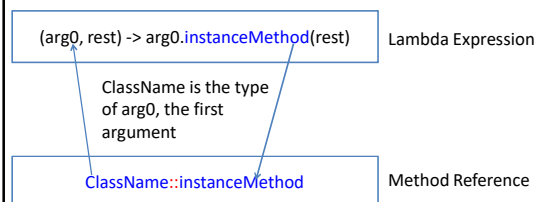
34

1. Method reference to a static method



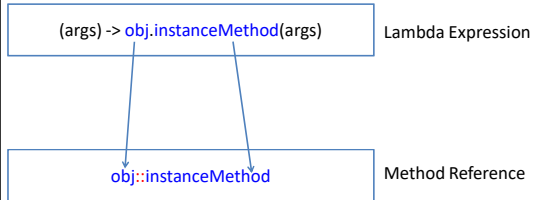
35

2. Method reference to an instance method



36

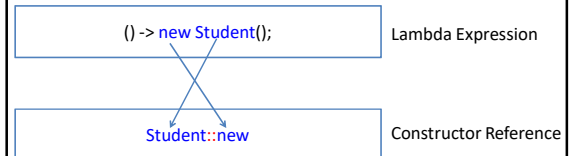
3. Method reference to an instance method of existing instance



37

Constructor References

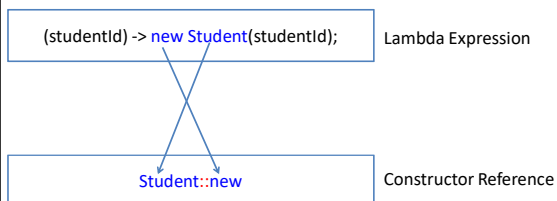
- We can create a reference to an existing constructor using its name and the keyword `new`
`ClassName::new`
- For example, reference to default constructor



38

Constructor References

- It is also possible to use non-default constructor as a method reference



39

References

- Java 8 Lambda Expressions & Streams, by Adib Saikali, Video, The San Francisco Java User Group
- Video lectures from JavaBrain series.

40