Lamda Expressions 101

# Lamda Expression

Lambda expressions enable you to treat functionality as method argument, or code as data.

Lamda expressions are converted into a private method of the enclosing class it is defined in by the Java compiler. It uses the “invokedynamic” instruction introduced as part of Java7, to bind this method dynamically.

Lamda are treated as

Basically, they are composed of an arguments section that can be empty, containing 0, 1 or more input parameters, an arrow (->) and a body that can be almost any kind of Java piece of code.

These are valid examples in pseudo code:

* (argument) -> { body } // just one input parameter
* (argument1, argument2...) -> { body } // more than one input parameter
* (x, y) -> x + y
* Arrays.asList( "1", "2", "3" ).forEach( e -> System.out.println( e ) );

## Example use case for lamda expression

Social application, search for persons matching a certain criteria. Possible approaches, in order of increasing efficiency

1. Create methods in Person, that searches for each matching param

New attributes will need to be included in the method. While creating generalized methods makes it better, this issue will still persist.

1. Create a search criteria code in a class/interface.

For new attributes, we will have to create/modify the class

1. Anonymous inner class for the search criteria
2. Use Lamda expressions

# Anonymous inner class vs Lamda Expressions

* 1. Code difference between anon and lamda expression

i. Anon class

printPersons(roster,

new CheckPerson() {

public boolean test(Person p) {

return p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25;

}

}

);

ii. Lamda expression

printPersons(roster,

**(Person p) -> p.getGender() == Person.Sex.MALE**

**&& p.getAge() >= 18**

**&& p.getAge() <= 25**);

Since, only one abstract method exists in a Functional interface, we do not need to specify the method name. In the above code

“(Person p)”: indicates the input param to the method.

* : points to the method’s body

“p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25” : Is the actual implantation of the method

As can been seen here, the lamda expression is passed a input to the method.

1. Differences between Anon class and Lamda expression is the resolution of the keyword “this”
   * 1. Anon class: this resolves to the anonymous class
     2. Lamda expression: this resolves to the enclosing class, where lamda is written
2. Scoping

The variables in lamda expressions are lexically scoped, thus they do not have any Shadowing issues.

# Structure of lamda expressions

* A lambda expression can have zero, one or more parameters.
* The type of the parameters can be explicitly declared or it can be inferred from the context. e.g. (int a) is same as just (a)
* Parameters are enclosed in parentheses and separated by commas. e.g. (a, b) or (int a, int b) or (String a, int b, float c)
* Empty parentheses are used to represent an empty set of parameters. e.g. () -> 42
* When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a\*a
* The body of the lambda expressions can contain zero, one or more statements.
* If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
* When there is more than one statement in body than these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

# Functional Interfaces

Similar to marker interfaces, we have functional interfaces, which are interfaces with one and exactly one abstract method. It can have more than one static/default methods.

Ex: java.lang.Runable, which has the run() method declared in it.

Java has some general purpose functional interfaces defined in java.lang.function

# Target typing

The target type for a lambda expression must be a functional interface and, to be compatible with the target type, the lambda expression must have the same parameter types as the interface’s function type, its return type must be compatible with the function type, and it can throw only exceptions allowed by the function type.

Two interfaces can use the same lamda expression, given the interface supports the lamda expressions input param and output.

# Method referencing

Java 8 allows for method referencing using the “::” . The below three snippets of code all do the same, even though they produce different bytecodes

// anon class

reduce(**new** IntBinaryOperator() {

**public** **int** applyAsInt(**int** left, **int** right) {

**return** Math.*max*(left, right);

}

});

// lamda expression

reduce( (**int** left, **int** right) -> Math.max(left, right) );

// method reference

reduce(Math::max);

1. A static method (ClassName::methName)
2. An instance method of a particular object (instanceRef::methName)
3. A super method of a particular object (super::methName)
4. An instance method of an arbitrary object of a particular type (ClassName::methName)
5. A class constructor reference (ClassName::new)
6. An array constructor reference (TypeName[]::new)

Refs:

<http://stackoverflow.com/questions/20001427/double-colon-operator-in-java-8>

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax>

<https://dzone.com/articles/java-lambda-expressions-basics>

<http://www.javaworld.com/article/2092260/java-se/java-programming-with-lambda-expressions.html>

<http://examples.javacodegeeks.com/java-basics/lambdas/java-8-lambda-expressions-tutorial/>

<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html>

<http://viralpatel.net/blogs/lambda-expressions-java-tutorial/>

<http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html>