Java 8 Lambda Expression Syntax

"Code with Passion!"

Topics

- What is and Why Lambda Expression (or simply Lambda)?
- Lambda implementation in Java
- What is Functional interface?
- Lambda expression syntax in Java 8
- Effectively final local variables

What is and Why Lambda Expression?

What is Lambda Expression (Lambda)?

- A formal system for “expressing computational behavior” (or “parameterizing behavior”)
  - Through functions (function objects)
- Function objects are first-class citizens
  - Function object can be assigned to a variable
  - Function object can be passed to a method as an argument
  - Function object can be returned as a return value
- Many modern programming languages support Lambda expression
  - JavaScript, List, Scheme, Ruby, Scala, Clojure, etc.
- Java 8 now supports Lambda expression
  - Biggest language change since Generics of Java SE 5

Why Lambda?

- Let you declare what to do, not how to do it
  - Cleaner, more concise, more expressive code
  - High productivity, flexible, “fluent” style programming is possible
- Promotes immutability
  - Less concurrency issues
- Enables parallel programming & lazy evaluation
  - Higher performance
- Forms the basis of functional programming paradigm
  - When functional programming is used, many set of problems are easier to solve, and results in cleaner code
- Richer collection APIs possible
  - Stream API

Java 8 Implementation of Lambda Expression

Lambda: Concept vs. Implementation

- Lambda expression is a concept
  - Different programming languages have different implementations of Lambda expression

- You, as a Java developer, need to learn both
  - General concept of Lambda expression and
  - How Java 8 implements Lambda expression

Java 8 Implementation of Lambda

- In Java, a Lambda expression is implemented essentially as an anonymous function
  - A Lambda expression is considered as an instance of a functional interface (an interface with a single abstract method)
  - The type of Lambda expression is indeed that functional interface

- There is no native “function” type (unlike in other languages), however, in Java 8 Lambda implementation
  - This is a deliberate decision by Java 8 Lambda designers

Usage Areas of Lambda in Java Programs

- Replacement of anonymous inner class
- Event handling
- Iteration over list
- Parallel processing of collection elements at the API level
- Functional programming
- Streams

Anonymous Inner Class Replaced by Lambda

Anonymous Inner Class and Lambda

- Given that typical usage of anonymous inner class is an example of an argument whose type is a functional interface, you can now replace it with a Lambda expression
  - In Java programs (of pre-Java 8 versions), anonymous inner class has been used as a kludge solution for passing a functional behavior (before Lambda is available in Java 8)

- Any code that uses Anonymous Inner class now can be simplified through the usage of Lambda
  - Just take the arguments and code block with following Lambda syntax removing everything else

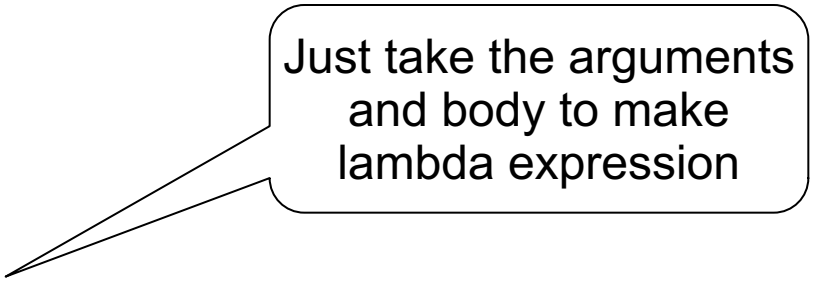
  (arguments) -> {code block}

Example #1: Runnable

- Anonymous Runnable replaced by Lambda

```
// Anonymous Runnable
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello world one!");
    }
};
r1.run();

// Lambda Runnable
Runnable r2 = () -> System.out.println("Hello world two!");
r2.run();
```



Example #2: ActionListener

- Anonymous ActionListener replaced by Lambda

```
// Anonymous ActionListener
testButton1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent event) {
        System.out.println("Click Detected by Anonymous Listener");
    }
});

// Lambda ActionListener
testButton2.addActionListener(event -> System.out.println("Click Detected by Lambda
    Listener"));
```

Just take arguments and body to make lambda expression

Example #3: Comparator

- Anonymous Comparator replaced by Lambda

Just take arguments and body to make lambda expression

```
// Anonymous Comparator
Collections.sort(personList, new Comparator<Person>(){
  public int compare(Person p1, Person p2){
    return p1.getSurName().compareTo(p2.getSurName());
  }
});

// Lambda Comparator
Collections.sort(personList, (Person p1, Person p2) →
                             p1.getSurName().compareTo(p2.getSurName()));
```

What is Functional Interface?

What is a Functional Interface (FI)?

- A regular Java interface with a single (abstract) method
  - It is common in Java programs
  - Sometimes called Single Abstract Method (SAM)

- Just like any other Java interface, it can be used as a reference type (type of a variable or type of an argument)
  - *MyFunctionalInterface x = (x,y) -> x+y ;*

- Even though it is a Java interface, it represents a function

FI is simply an Interface with a single method

- In fact, previous versions of Java (Java 7 and earlier versions) have several functional interfaces already

```
// Runnable interface
public interface Runnable {
    public abstract void run();
}

// ActionListener interface
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}

// Comparator interface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Backward Compatibility

- Any interface with a single method is considered as a functional interface by Java 8+
- Java 8+ Lambda works with old libraries that use functional interfaces without any need to recompile or modification of them

@FunctionalInterface Annotation

- When used, Java 8 compiler produces an error if the interface has more than one method - helps developers at compile time (just like @Override annotation helps developers find at compile time incorrect method name)

```
@FunctionalInterface
public interface MyInterface {
    public String  myMethod();
}

// Generates Invalid @FunctionalInterface compile error
@FunctionalInterface
public interface MyInterface {
    public String  myMethod();
    public String  myMethod2();
}
```

Where to use Lambda Expression in Java app?

- Concept
  - You use Lambda expression wherever a functional behavior is required
- Java app
  - You can use Lambda expression in any place where the functional interface type is expected
- Examples
  - You can assign a lambda expression to a variable whose type is a functional interface
  - You can pass a lambda expression to a method as an argument whose type is a functional interface

Example #1: Variable is functional interface type

- Let's say we have a functional interface

```
@FunctionalInterface
public interface Calculator {
   int calculate(int x, int y);
}
```

- A variable whose type is a functional interface can be assigned with a lambda expression

```
Calculator multiply = (x,y) -> x*y;
Calculator divide = (x,y) ->x/y;
int product = multiply.calculate(50,10);
int quotient = divide.calculate(50,10);
```

Example #2: An argument is functional interface

- Let's say we have a functional interface (same as in prev. slide)

```
@FunctionalInterface
public interface Calculator {
   int calculate(int x, int y);
}
```

- Types of arguments are functional interface

```
public static void myMethod(Calculator m, Calculator d){
    int product = m.calculate(60, 10);
    int quotient = d.calculate(60, 10);
    System.out.println("product = " + product + " quotient = " + quotient);
}
```

- Pass lambda expressions as arguments of a method

```
myMethod((x,y)->x+y, (x,y)->x/y);
```

Lambda Expression Syntax in Java

Lambda Expression Syntax

- General syntax
  - (argument list) -> { code block}

- Syntax can be simplified in the following ways
  - #1: Type inferencing for the arguments
  - #2: Omitting parentheses for a single argument
  - #3: When a body has only a single expression
    - (1) no need to use return
    - (2) no need to use semi-colon
    - (3) no need to use curly braces {..}

#1: Type inferencing for the arguments

- Types in argument list can be omitted
  - Java compiler already knows the types of the arguments from the single method signature of the functional interface of the lambda expression

```
// Instead of this
(String myArg1, Integer myArg2) → {... }

// You can do this because types of the arguments can be inferred by the compiler
(myArg1, myArg2) → {… }
```

#2: Single argument with no ( )

- If there is a single argument, parentheses ( ) are optional

```
// Instead of this
(myArg1) → {... }

// You can do this because there is a single argument
myArg1 → {… }
```

#3: When body has only a single expression

- When the body (code block) has only a single expression, the value of the expression automatically becomes a return value
  - No need to specify return statement
  - No need to use semi-colon at the end
  - No need to enclose the expression with { }

- If the body has multi-line code, then no simplification is allowed

```
// Instead of this
(myArg1, myArg2)  → { return (someExpression); }

// You can do this because the body has only a single expression
(myArg1, myArg2)  → someExpression
```

Simplification Examples of Lambda Expression

```
(int x, int y) -> { return x+y;}
        (x,y) -> { return x+y;}
        (x,y) -> x+y
           x  -> x*2

() ->  System.out.println("Hello, world!")

x -> { System.out.println(x);
       System.out.println(x*2);
       return x*2; }
```

You must use curly braces { } and use return statement because there are multiple statements

Effectively Final Local Variables

Effectively Final Local variables

- The local variables in the enclosing body can be accessible to Lambda expression
- Lambda expression, however, cannot change them - they are behaving like final variables – hence the reason why they are called “effectively final local variables”

```
public class Main {

  public static void main(String[] args) {

    int myInt = 100;
    Calculator multiply = (int x, int y) -> {return x * y * myInt;};
    Calculator divide = (int x, int y) -> {myInt=30; return x * y * myInt;}; // compile error
  }

}
```

Code with Passion!