



# Lambda Expressions in Java

Pasha Ranakusuma, Thejus Unniveelan, Omar  
Elgeoushy, Mohammed Alsabty, Sam Lightfoot



# Overview

- Lambda Expressions : Methods/Functions :: Object Oriented Programming : Objects
- Expression vs Statement
- Helps make code that is more dynamic in their capabilities

```
public static void main(String[] args) {  
    ArrayList<Integer> numbers = new ArrayList<Integer>();  
    numbers.add(5);  
    numbers.add(9);  
    numbers.add(8);  
    numbers.add(1);  
    numbers.forEach( (n) -> { System.out.println(n); } );  
}
```



## History and Development

- In 1936 lambda calculus was developed by the american mathematician Alonzo Church
- Lambda was used as a model to solve the Entscheidungsproblem (first order logic)
- Functional programming languages, like Miranda, ML etcetera, are based on the lambda calculus.





## The Case For and Against Lambda Expressions


Used in place of **inner anonymous classes** to represent **functional interfaces**.

### Pros:

- Concise
- Loaded quickly without creating classes
- Works in conjunction with Java Stream to function concurrently

### Cons:

- Counters Java conventions
- Cold start executions



## Similarities/Differences in Other Notable Languages

- Present in nearly every single modern languages including Python, C, C++, etc.
- Most things are the same as in Java:
  - - An argument list: Zero or more **variables** separated with commas, usually in parentheses. C#, Java, and Javascript allow omitting parentheses if there is only one argument. C++ requires the arguments to have types, but the other statically typed languages allow it without making it required.
    - A body: Statements to execute or an expression to evaluate and return. When using statements, they are surrounded with braces, and a **return** statement can be used to return a value. When the body is just an expression, braces are omitted. (C++ only allows using statements and Python only allows using expressions. In C#, braces can be omitted when the body is a single statement.)
- But there are syntactical and functional differences in different languages.

#### Maximum value, in C, C++, C#, and Java

```
double max(double x, double y) {  
    if (x > y)  
        return x;  
    return y;  
}
```

and then as a lambda expression in various languages:

#### C++

```
[](double x, double y) {  
    if (x > y)  
        return x;  
    return y;  
}
```

#### C# and Javascript

```
(x, y) => {  
    if (x > y)  
        return x;  
    return y;  
}
```

#### Java

```
(x, y) -> {  
    if (x > y)  
        return x;  
    return y;  
}
```

#### Erlang [\[edit\]](#)

Erlang uses a syntax for anonymous functions similar to that of named functions.<sup>[16]</sup>

```
% Anonymous function bound to the Square variable  
Square = fun(X) -> X * X end.  
  
% Named function with the same functionality  
square(X) -> X * X.
```

#### Go [\[edit\]](#)

Go supports anonymous functions.<sup>[21]</sup>

```
foo := func(x int) int {  
    return x * x  
}  
fmt.Println(foo(10))
```

#### Haskell [\[edit\]](#)

Haskell uses a concise syntax for anonymous functions (lambda expressions). The backslash is supposed to resemble  $\lambda$ .

```
\x -> x * x
```

Lambda expressions are fully integrated with the type inference engine, and support all the syntax and features of "ordinary" functions

```
map (\x -> x * x) [1..5] -- returns [1, 4, 9, 16, 25]
```

The following are all equivalent:

```
f x y = x + y  
f x = \y -> x + y  
f = \x y -> x + y
```



## Functional differences:

- Statically typed vs dynamically typed
  - E.g. Python can assume return types while others may not
- Use of pointers in languages which allow it
- In javascript, asynchronous vs synchronous lambda expressions

```
JS sequential.js ●
1
2 // I didn't mean this!
3
4 exports.handler = async (event) => {
5   const restrictedList = await checkRestrictedList()
6   const accBalance = await checkAccountBalance()
7   const marketOpen = await isMarketOpen()
8
9   const allowed = (!restrictedList && accBalance > 0 && marketOpen) ? true : false
10
11   return {
12     statusCode: 200,
13     allowed,
14     body: JSON.stringify('This took forever. Urgh')
15   }
16 }
17
18
```



# Coding Examples - Interfaces

```
/**
 * Specific void Functional Interface
 *
 * Designed only to run a zero-parameter Lambda function with a void return
 * value
 */
interface EmptyFunction {
    void run();
}

/**
 * Specific String Functional Interface
 *
 * Designed only to run a one-parameter Lambda function with a String return
 * value
 */
interface StringFunction {
    String run(String str);
}

/**
 * Specific void Functional Interface
 *
 * Designed only to run a two-parameter Lambda function with an int return
 * value
 */
interface DualFunction {
    int run(int a, int b);
}
```





# Coding Examples - Sample Expressions

```
// Simple case of a void-returning zero-parameter Lambda Expression
EmptyFunction empty = () -> {
    System.out.println("No parameters here!");
};

// Simple case of a String-returning one-parameter Lambda Expression
StringFunction writeMessage = (String message) -> {
    return "The message is as follows:\n" + message;
};

// Simple case of a two-parameter Lambda Expression
DualFunction multiplication = (int a, int b) -> {
    return a * b;
};
```

# Coding Examples - Output

```
switch (args[0]) {  
  
    // Use of zero-parameter Lambda Expression  
    case "0":  
        empty.run();  
        break;  
  
    // Use of single-parameter Lambda Expression  
    case "1":  
        String message = writeMessage.run("Tada! This is the result of  
        another Lambda Expression");  
        message = exclaim.run(message);  
        System.out.println(message);  
        break;  
  
    // Use of two-parameter Lambda Expression  
    case "2":  
        int a = 99;  
        int b = 101;  
        int product = multiplication.run(a, b);  
        System.out.println("The product of " + a + " and " + b + " is "  
        + product);  
        break;  
}
```

No parameters here!

The message is as follows:

Tada! This is the result of another Lambda Expression!

The product of 99 and 101 is 9999



## Sources for further reading

<https://joshdata.me/lambda-expressions.html#:~:text=A%20guide%20to%20programming%20lambda,languages%20for%20writing%20short%20functions.>

<https://www.jrebel.com/blog/pros-and-cons-of-lambdas-in-java-8>

<http://worldcomp-proceedings.com/proc/p2015/SER2509.pdf>

GitHub Repo for some Lambda Expressions:

<https://github.com/Bey2001/Lambda-Expressions>



## Potential Questions?

How do you feel about functional programming?

Are you now more likely to use lambda expressions in your work?