Lambda Functions in Java

Giannis Kalopisis

M135 - Advanced Programming Methods

What is Lambda?

What is Lambda (λ)?

1. The 11th letter of the Greek alphabet (**λ**) - Alphabet

2. The symbol for **wavelength** in mathematical equations - Mathematics

3. An **anonymous functions** - Computer Science

A little bit of history

Alonso Church
1903 - 1995

- Mathematician at Princeton University
- Interests: **Notion of a function from a computational perspective**

His answer to that?

Lambda Calculus

A little bit of history

Alan Turing
1912 - 1954

- Mathematician & Computer Scientist at Princeton University
- Inventor of Turing Machine
- Turing Machines capture a basic **state-based model of computation**

A little bit of history

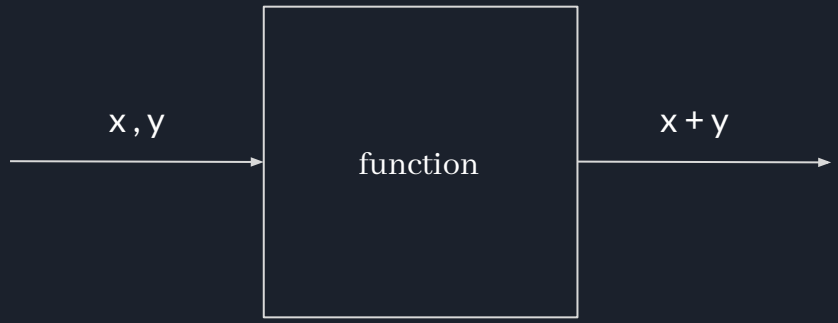
Basic **Functional** notion of computation
(Lambda Calculus)

Church - Turing Hypothesis

Basic **State-Based** notion of computation
(Turing Machine)

- Church - Turing Hypothesis: Functional & State-Based notion of computation are **EQUIVALENT**

Lambda Calculus - Church



Function is a black box:

- Takes input
- Gives output

Lambda Calculus - Church

Important details:

- They are **black boxes**
  - Can't see the internal mechanics
  - Can only give input and observe the output

- They are **pure mathematical functions** with **no internal state**
  - Map input and output
  - No internal state or hidden information to use
  - Completely opposite notion of computation than Turing Machines (based on internal-states)

Lambda Calculus - Church

Pure mathematical functions:

- ***Given the same inputs, a function will always produce the same output*** and will not cause any side effects. Its behavior is solely determined by its arguments.

No internal state:

- ***Functions don't maintain any internal state or store any information*** beyond what is explicitly provided in their arguments.

And some maths...

Lambda Calculus syntax (addition function):

- Parameter1: λx
- Parameter2: λy
- Output: x + y

$$\lambda x.\ \lambda y.\ x + y$$

What's the point?

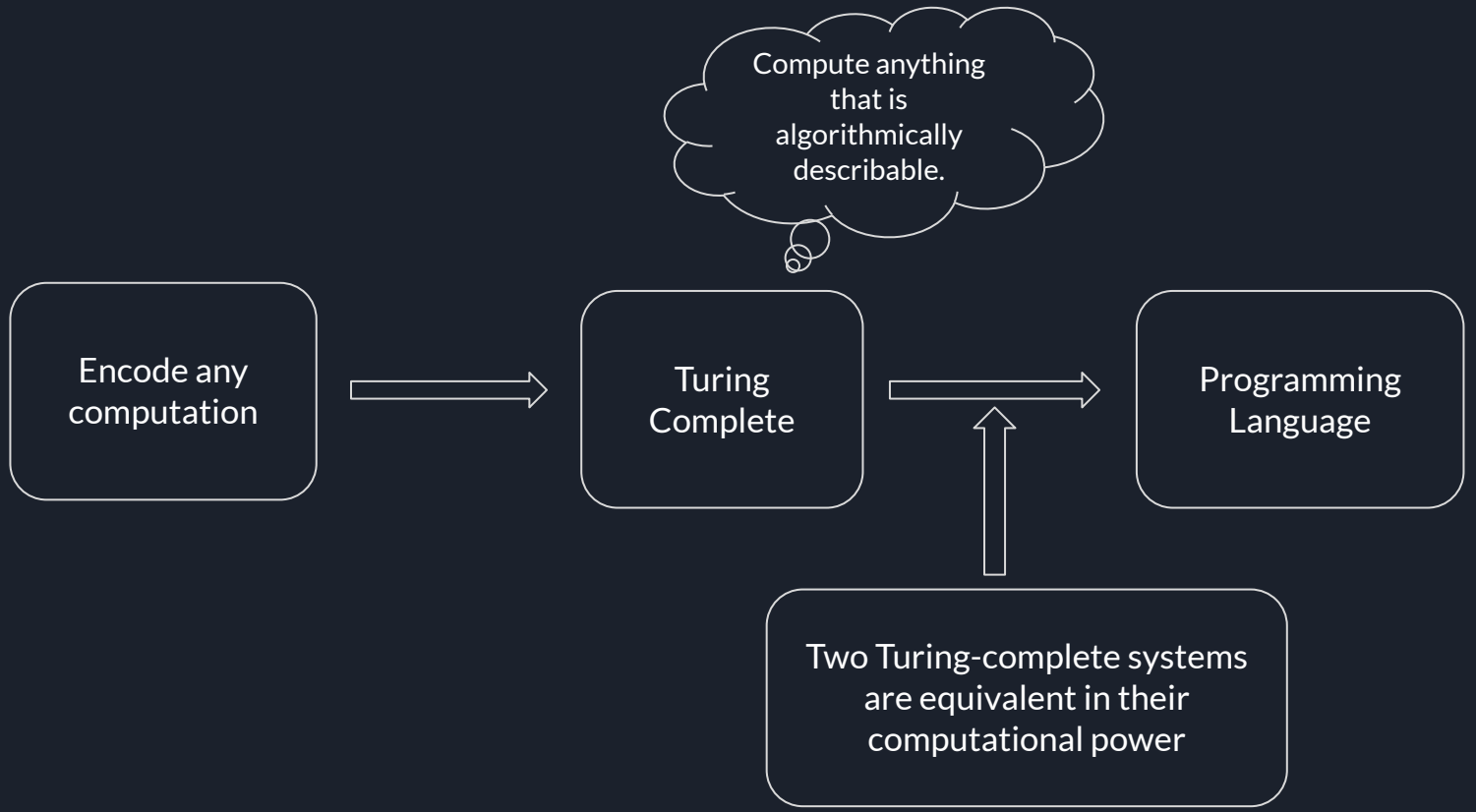
1. Can encode <u>any</u> computation
    a. A program from any programming language (invented or ever will be invented) it can be encoded to Lambda Calculus
    b. So, what kind of program you can encode? ANYTHING

Any Turing Machine program can be translated into an **equivalent Lambda Calculus program**

What's the point?

1. Can encode any computation
    a. A program from any programming language (invented or ever will be invented) it can be encoded to Lambda Calculus
    b. So, what kind of program you can encode? ANYTHING
2. Basis for functional programming
    a. Haskell: compiled down to a very small core language, essentially a glorified form of Lambda Calculus
    b. ML family
3. Now in most languages
    a. Python, Java, C#, etc.

Completeness



Lambda Calculus & Lambda Functions

Why?

Conceptual Similarities:

1. **Abstraction**: Both allow the definition of anonymous functions
2. **Higher-Order Functions**: Both support higher-order functions
    a. Enable functions to be passed as arguments or returned from other functions
3. **Syntax**: The syntax of lambda functions in languages is directly inspired by lambda calculus.
    a. Lambda Calculus: λx . x
    b. Python: lambda x: x
4. **Functional Programming Paradigm**: Shape functional programming languages
    a. Haskell, Lisp, Scheme

Lambdas in Java

Syntax

Simple syntax with **one parameter**:

- parameter -> expression

More parameters (use parentheses):

- (parameter1, parameter2) -> expression

No parameters:

- () -> expression

Syntax

Any problems with those syntaxes?

- They have to immediately return a value
- They cannot contain variables, assignments or statements

Solution:

- (parameter1, parameter2) -> { code block }

Syntax Examples

Example with no parameters:

```
// Example with no parameters
// Syntax: () -> expression
Supplier<String> greeting = () -> "Hello, World!";
System.out.println(greeting.get()); // Output: Hello, World!
```

Example with one parameters:

```
// Example with a single parameter
// Syntax: (parameter) -> expression
Function<Integer, Integer> square = (x) -> x * x;
System.out.println(square.apply( t: 5)); // Output: 25
```

Syntax Examples

Example with two parameters:

```
// Example with multiple parameters
// Syntax: (parameter1, parameter2) -> expression
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
System.out.println(add.apply( t: 3,  u: 4)); // Output: 7
```

Example with block of code:

```
// Example with a block of code
// Syntax: (parameters) -> { statements; }
Consumer<String> printUpperCase = (str) -> {
    String upperCase = str.toUpperCase();
    System.out.println(upperCase);
};
printUpperCase.accept( t: "hello"); // Output: HELLO
```

Method Reference

Method reference:

- Method references provide ***a way to refer to methods without invoking them***
- Replace lambda expressions when the lambda's sole purpose is to call a method

```
// Using lambda expression
Consumer<String> printLambda = (str) -> print(str);
printLambda.accept( t: "Hello Lambda"); // Output: Hello Lambda
```

```
// Using method reference
Consumer<String> printMethodRef = Main::print;
printMethodRef.accept( t: "Hello Method Reference"); // Output: Hello Method Reference
```

- The syntax Main::println is equivalent to a lambda expression that takes an argument and passes it to my print method of Main class.

Lambdas internal implementation

Functional Interfaces

Definition:

- Functional interfaces are Java interfaces that contain **exactly one abstract method**.
  - Can also have default or static methods.

Purpose:

- Introduced in Java 8 to support functional programming and lambda expressions.
  - A concise way to express instances of single-method interfaces
- Treat ***functionality as a first-class citizen***, allowing ***methods to be passed around*** as if they were data.

Annotation:

- Often annotated with the **@FunctionalInterface** annotation.

Functional Interfaces

A **functional interface** is nothing but an **interface that has only one abstract method**.

Lambda expressions are nothing but the **implementation of the functional interface**.

Functional Interfaces

Example:

```
1 usage  new*
@FunctionalInterface
interface TriFunction<T, U, V, R> {
    1 usage  new*
    R apply(T t, U u, V v);
}

1 usage  new*
@FunctionalInterface
interface MyFunctionalInterface {
    1 usage  new*
    void myMethod();
}
```

```
public class Main {
    new*
    public static void main(String[] args) {
        TriFunction<Integer, Integer, Integer, Integer> add = (a, b, c) -> a + b + c;

        int result = add.apply( t: 2,  u: 3,  v: 4);
        MyFunctionalInterface myFunctionalInterface = () -> System.out.println(result);
        myFunctionalInterface.myMethod();
    }
}
```

Common Functional Interfaces

Java provides a set of functional interfaces in the `**java.util.function**` package, such as:

- Consumer
- Function
- Predicate
- ...

which are **commonly used in functional programming with lambda expressions**.

These interfaces **cover various use cases** and allow developers to write more concise and expressive code.

Common Functional Interfaces - Examples

1. **Function<V, T>**: Takes one argument of type **V** and produces a result of type **T**

```
Function<String, Integer> transformLength = str -> str.length();
```

2. **BiFunction<V, V, T>**: Takes two arguments of type V, produces a result of type T

```
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
```

3. **Supplier<T>**: Represents a supplier with no arguments, produces a result of type **T**

```
Supplier<Double> getRandom = () -> Math.random();
```

4. **Consumer<V>**: Takes a single argument of type **V** and returns no result

```
Consumer<String> printUpperCase = str -> System.out.println(str.toUpperCase());
```

Common Functional Interfaces - Examples

5. **Predicate<V>**: Represents a boolean-valued function of one argument of type **V**

```
Predicate<Integer> isPositive = num -> num > 0;
```

6. **BiPredicate<V, T>**: Represents a predicate of two arguments of types **V and T**

```
BiPredicate<String, Integer> hasSameLength = (s, len) -> s.length() == len;
```

5. **UnaryOperator<V>**: An operation on a single operand of type **V**, producing a result of the same type **V**

```
UnaryOperator<Integer> square = num -> num * num;
```

8. **BinaryOperator<V>**: An operation upon two operands of the same type **V**, producing a result of the same type **V**

```
BinaryOperator<Integer> multiply = (a, b) -> a * b;
```

Actual code example

```
public void findPackagesPerModule() {
    this.moduleList.forEach( m -> {
                Set<String> packages = m.getPackages() Set<String>
                        .stream() Stream<String>
                        .filter(packageName -> packageName.startsWith("java.") || packageName.startsWith("javax."))
                        .collect(Collectors.toSet());
                this.packagesPerModuleMap.put(m, packages);
            });
}
```

This lambda doesn't explicitly return a value (it performs actions and doesn't have a return statement), it's transforming or processing each m element. Therefore, in terms of functional programming, this lambda could be seen as ***Consumer***.

Lambdas power

Why use Lambdas?

Conciseness and Readability

Lambdas can ***reduce verbosity*** and ***enhance readability*** in code by providing a concise way to define simple functionalities.

1. **Conciseness:** express functionality in a single line, reducing the need for a separate named function.
2. **Inline Usage:** used directly where they're needed, eliminating the need for a named function that might only be used in one specific context.
3. **Avoiding Overhead:** In cases where a function is simple and used only once, creating a named function might add unnecessary overhead.

Conciseness and Readability

Without Lambdas:

```java
public class WithoutLambdaExample {
    new *
    public static void main(String[] args) {
        List<Integer> integerList = new ArrayList<>();
        integerList.add(1);
        integerList.add(2);
        integerList.add(3);

        // Without using lambdas
        for (Integer num : integerList) {
            System.out.println(num);
        }
    }
}
```

With Lambdas:

```java
public class WithLambdaExample {
    new *
    public static void main(String[] args) {
        List<Integer> integerList = new ArrayList<>();
        integerList.add(1);
        integerList.add(2);
        integerList.add(3);

        // Using lambdas for concise iteration and printing
        integerList.forEach(m -> System.out.println(m));
    }
}
```

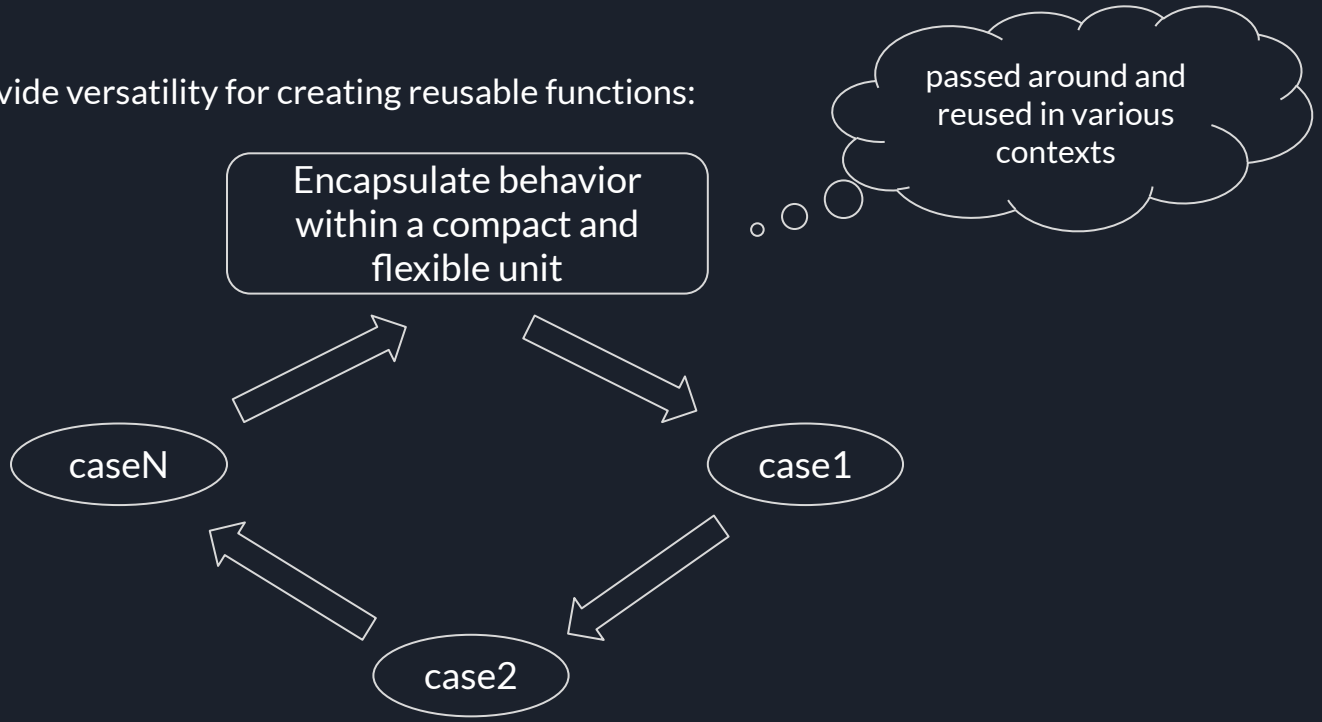
Functional Programming Paradigm

Lambdas encourage functional programming principles by enabling the **use of functions as first-class citizens**.

1. **Functions as Values:** Lambdas allow functions to be treated as values, enabling them to be passed as arguments to other functions, returned as results, or stored in data structures (treating functions as any other data type.)
2. **Higher-Order Functions:** Functions that either take other functions as arguments, return functions as results, or both (composing complex functionalities by combining simpler functions, promoting modularity and reusability).
    a. Enhance Streams API, encourages the use of higher-order functions like map, filter, reduce.
3. **Declarative Style:** Lambdas enable a more declarative style of programming, expressing the "what" rather than the "how".
4. **Pure Functions:** Encourage writing pure functions (functions with no side effects)

Flexibility and Code Reusability

Lambdas provide versatility for creating reusable functions:



Flexibility and Code Reusability

```java
public static void main(String[] args) {
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie",
            "David", "Eve", "Anna");

    // Reusable filter functions using lambdas
    Predicate<String> startsWithA = name -> name.startsWith("A");
    Predicate<String> hasLengthFive = name -> name.length() == 5;

    List<String> filteredNamesA = filterNames(names, startsWithA);
    List<String> filteredNamesLength = filterNames(names, hasLengthFive);

    System.out.println("Names starting with 'A': " + filteredNamesA);
    System.out.println("Names with length 5: " + filteredNamesLength);
}
```

```java
public static List<String> filterNames(List<String> names,
                                       Predicate<String> condition) {
    List<String> filteredNames = new ArrayList<>();
    for (String name : names) {
        if (condition.test(name)) {
            filteredNames.add(name);
        }
    }
    return filteredNames;
}
```

Streams + Lambdas = <3

Streamlining APIs and Iteration

Iterate through items:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Using forEach to iterate and print elements
numbers.forEach(number -> System.out.println(number));
```

Make a collection of items stream and apply functions on it:

```
List<String> names = Arrays.asList("Alice", "Bob",
        "Charlie", "David", "Eve");

// Filtering names starting with 'A'
names.stream()
        .filter(name -> name.startsWith("A"))
        .forEach(System.out::println);
```

Convert a collection to stream and apply higher-order functions:

```
List<String> result = names.stream()
        .filter(str -> str.length() > 5)
        .map(String::toUpperCase)
        .toList();

result.forEach(System.out::println);
```

Parallel Processing and Multithreading

1. **Facilitating Parallel Processing**
   a. **Simplified Parallelization:** Simplify the usage of concurrent APIs by encapsulating tasks as functional units.
   b. **Parallel Stream Operations:** Lambdas, when combined with the Streams API, allow for easy parallelism by leveraging parallel stream operations (**parallelStream()**). These operations automatically distribute tasks across multiple threads, allowing for parallel processing of collections.

```
ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4);

// Task encapsulation using lambdas for ExecutorService
for (int i = 0; i < 10; i++) {
    int taskNumber = i;
    executor.submit(() ->
            System.out.println(taskNumber + " thread: " +
                    Thread.currentThread().getName()));
}

executor.shutdown();
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6,
                                      7, 8, 9, 10);

// Parallel processing using lambda in parallelStream()
int sum = numbers.parallelStream() Stream<Integer>
        .mapToInt(i -> i) IntStream
        .sum();
System.out.println("Sum of numbers: " + sum);
```

Parallel Processing and Multithreading

2. **Functional Approach to Multithreading**
   a. **Encapsulation of Tasks:** Lambdas encapsulate tasks or operations, enabling them to be easily passed to multithreading constructs like ExecutorService, Thread.
   b. **Simplified Thread Creation:** Lambdas can be used as concise **implementations of the Runnable or Callable interfaces**, simplifying the creation of threads or tasks in multithreaded environments. Simplify the creation of a Callable instance, ***avoiding the need for a separate class or implementing a verbose anonymous inner class.***

```
// Task encapsulation using lambdas for Runnable interface
Runnable task = () -> {
    System.out.println("Task executed by thread: "
            + Thread.currentThread().getName());
};

Thread thread = new Thread(task);
thread.start();
```

```
// Task encapsulation using lambdas for Callable interface
Callable<Integer> task = () -> 42;

// Creating a FutureTask using the Callable task
FutureTask<Integer> futureTask = new FutureTask<>(task);

// Creating a Thread with the FutureTask and starting it
new Thread(futureTask).start();

// Getting the result from the FutureTask
int result = futureTask.get();
System.out.println("Task result: " + result);
```

Problems?!

Problems of Lambdas in programming

While lambda functions offer numerous advantages, they come with certain drawbacks that need careful consideration:

1. Complexity and Readability
2. Debugging Challenges
3. Learning Curve and Compatibility

Complexity and Readability

Nested or intricate lambda expressions can make code harder to comprehend, especially for developers unfamiliar with functional programming concepts.

```
this.moduleList.forEach(m -> {
    Optional<ResolvedModule> resolved = this.bootConfig.findModule(m.getName());
    resolved.ifPresent(rm -> {
        ModuleReference ref = rm.reference();
        try (ModuleReader reader = ref.open()) {
            reader.list().forEach(s -> {
                try {
                    if (s.endsWith(".class") && !s.equals("module-info.class")) { // exclude non-class resources & the module-in
                        String packageName = s.substring(0, s.lastIndexOf( ch: '/')).replace( oldChar: '/', newChar: '.');
                        if (m.isExported(packageName)) {
                            String className = s.replace( oldChar: '/', newChar: '.').substring(0, s.length() - ".class".length());
                            // Add the class to the existing map
                            this.types.add(Class.forName(className));
                        }
                    }
                } catch (Exception ex) {
                    System.out.println(ex.getMessage());
                    ex.printStackTrace(System.out);
                }
            });
        } catch (IOException e) {
            System.out.println(e.getMessage());
            e.printStackTrace(System.out);
        }
    });
});
```

Debugging Challenges

- Understanding the flow and behavior of complex lambda expressions during debugging sessions can be daunting, affecting development and troubleshooting processes.

- Lambdas, especially those with nested or intricate logic, might obscure the sequence of operations and make it challenging to isolate and rectify issues.

Learning Curve and Compatibility

Lambda expressions often introduce new concepts. Understanding and effectively utilizing lambdas might require:

- learning new syntax
- grasping functional programming principles:
  - Higher-order functions
  - Immutability
  - Closures

Additionally, compatibility issues might arise when working with older systems or languages that do not support lambda expressions or have limited support for functional programming features.

The impact of Lambdas in performance

Performance Overhead

Using lambda expressions `***might***` introduce a slight performance impact compared to equivalent non-lambda code.

```
private static HashMap<Integer, Integer> map = new HashMap<>();

1 usage  new *
private static int max100(int i) {
    var result = map.get(i);
    if (result == null) {
        map.put(i, 100);
        return 100;
    }
    return result;
}
```

```
private static HashMap<Integer, Integer> map = new HashMap<>();

1 usage  new *
private static int max100(int i) {
    return Optional.ofNullable(map.get(i))
            .orElseGet(() ->
            { map.put(i, 100); return 100; });
}
```

In this scenario, the lambda expression simplifies the code and makes it more concise.

However, internally, the lambda expression involves the creation of an additional object (an instance of a functional interface) to represent the lambda, which might incur a small performance cost compared to the non-lambda approach.

Performance dependency

What does performance depend on?

- Your code:
  - Algorithmic complexity
  - Produced bytecode
- JVM implementation
- Running phase:
  - Cold
  - Hot

Improve performance

1. The JIT compiler optimizes at the method level. It only optimizes the most promising methods. That's methods that have run hot by being called again and again, and short methods are compiled earlier. The longer a method gets, the less likely it's optimized.

2. Performance impact of functional programming it's there, but almost exclusively during the cold start phase. The JIT compiler optimizes most of the overhead.

3. Real-world programs usually involve slow stuff like a database or file access, so you're free to choose your preferred approach without worrying about performance.

Links for the performance impact

Links for more on performance:

- https://www.beyondjava.net/functional-programming-java-performance-impact
- https://www.beyondjava.net/blog/performance-java-8-lambdas
- https://stackoverflow.com/questions/27524445/does-a-lambda-expression-create-an-object-on-the-heap-every-time-its-executed
- https://stackoverflow.com/questions/16827262/how-will-java-lambda-functions-be-compiled

Conclusion

Conclusion

- Lambda Calculus, a mathematical concept of computation, introduced us to functional programming and Lambda functions.
- Lambdas in Java have revolutionized programming by introducing functional paradigms through functional interfaces.
- As we see every day in our science, many ideas combine to improve each other.
- And finally, as with any `new` idea there are not only advantages, and therefore we should be careful.

Conclusion


Let's embrace their power while navigating their challenges, paving the way for more efficient and expressive code.
ChatGPT