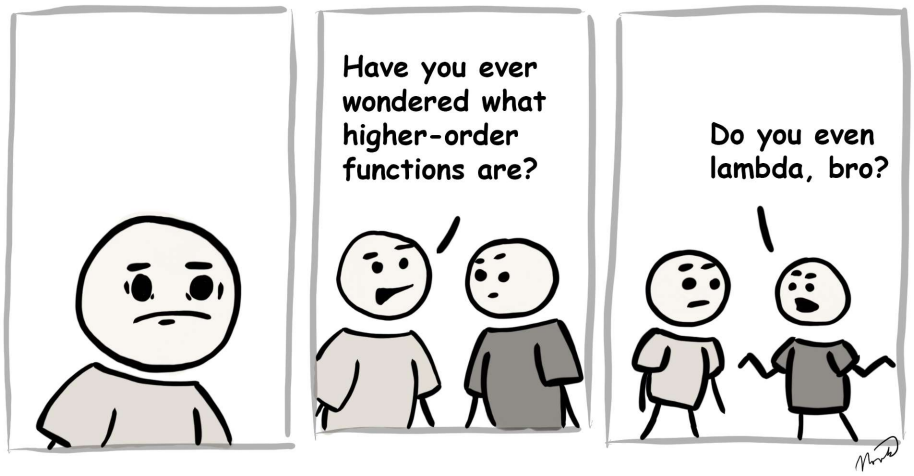

OBJECT-ORIENTED PROGRAMMING

Lambda Expressions

Lecture #8

doc. Ing. Martin Tomášek, PhD.
Departments of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2024/2025

What is functional programming?

- A style of programming that treats computation as the evaluation of mathematical functions
  - Programs are constructed by applying and composing functions
- Declarative programming paradigm
  - Expressions defining a function, rather than a sequence of imperative statements which change the state of the program
- High-order functions – functions can take functions as arguments and return functions as results
- Pure functions – eliminates side effects
- Expressions have referential transparency
- Treats data as being immutable
- Prefers recursion over explicit loops
- Functional programming languages: Lisp, Haskell, Ocaml, F#, etc.

What do functional programming?

- Allows us to write easier-to-understand, more declarative, more concise programs than imperative programming
- Allows us to focus on the problem rather than the code
- Facilitates parallelism
- Example: Factorial

Imperative language (C):

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  return n * factorial(n - 1);
}
```

Functional language (Haskell):

```
factorial 0 = 1
factorial n = n * factorial(n - 1)
```

Composition of functions

Function composition in functional language (Curry):

```
const compose = f => g => x => f(g(x));

---

compose (x => x * 4) (x => x + 3) (2);
// (2 + 3) * 4
// => 20
```

Function composition in imperative language (Javascript):

```
let compose = function(f, g) {
  return (x) => f(g(x));
}
```

or

```
let compose = (f, g) => (x) => f(g(x));

---

compose((x) => x * 4, (x) => x + 3)(2);
// (2 + 3) * 4
// => 20
```


The lambda calculus

- Formal model of computation underlying all functional programming languages
- Introduced in the 1930s by Alonzo Church as a mathematical system for defining computable functions
- Lambda calculus and Turing machines are equivalent models of computation (showing that the lambda calculus is Turing complete)
  - Basis for Church-Turing thesis
- Features from the lambda calculus such as lambda expressions have been incorporated into many widely used programming languages like C++, C#, Java, Python, Javascript, etc.

What is the lambda calculus?

- The central concept in the lambda calculus is an expression generated by the following grammar which can denote a function definition, function application, variable, or parenthesized expression

  $E ::= \lambda\, x\, .\, E \mid E\ E \mid x \mid (E)$

- We can think of a lambda-calculus expression as a program which when evaluated by beta-reductions

  $((\lambda\, x\, .\, E)\ E') \rightarrow E\{x \leftarrow E'\}$

  returns a result consisting of another lambda-calculus expression

Example of a lambda expression

- The lambda expression

  $\lambda\, x\, .\, (x + 1)\ 2$

  represents the application of a function $\lambda\, x\, .\, (x + 1)$ with a formal parameter x and a body $x + 1$ to the argument 2
- Execution of the expression (beta-reduction is applied)

  $\lambda\, x\, .\, (x + 1)\ 2 \rightarrow (x + 1)\{x \leftarrow 2\} \rightarrow 2 + 1$
- Notice that the function definition $\lambda\, x\, .\, (x + 1)$ has no name – it is an anonymous function
- In Java, we would represent this function definition by the Java lambda expression

```
x -> x + 1
```

Examples of Java lambda expressions

- A Java lambda is basically a method in Java without a declaration usually written as

```
(parameters) -> { body }
```

- Examples

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

- A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context
- Parenthesis are not needed around a single parameter
- `()` is used to denote zero parameters
- The body can contain zero or more statements
- Braces are not needed around a single-statement body

Benefits of lambda expressions in Java

- Enabling functional programming
- Writing leaner more compact code
- Facilitating parallel programming
- **Developing more generic, flexible and reusable APIs**
- Being able to pass behaviors as well as data to functions

Example: Print a list of integers

```
List<Integer> intSeq = Arrays.asList(1, 2, 3);
intSeq.forEach(x -> System.out.println(x));
```

- `x -> System.out.println(x)` is a lambda expression that defines an anonymous function with one parameter named `x` of type `Integer`

Example: Multiline lambda

```
List<Integer> intSeq = Arrays.asList(1, 2, 3);
intSeq.forEach(x -> {
   x += 2;
   System.out.println(x);
});
```

- Braces are needed to enclose a multiline body in a lambda expression

Example: Lambda with defined local variable

```
List<Integer> intSeq = Arrays.asList(1, 2, 3);
intSeq.forEach(x -> {
   int y = x * 2;
   System.out.println(y);
});
```

- Just as with ordinary functions, you can define local variables inside the body of a lambda expression

Example: Lambda with a declared parameter type

```
List<Integer> intSeq = Arrays.asList(1, 2, 3);
intSeq.forEach((Integer x) -> {
   x += 2;
   System.out.println(x);
});
```

- You can, if you wish, specify the parameter type

Implementation of Java lambda

- The Java compiler first converts a lambda expression into a function
- It then calls the generated function
- For example, `x -> System.out.println(x)` could be converted into a generated static function

```
public static void genName(Integer x) {
    System.out.println(x);
}
```

- But what type should be generated for this function? How should it be called? What class should it go in?

Functional interfaces

- Design decision: **Java lambdas are assigned to functional interfaces**
- A functional interface is a Java interface with exactly one non-default method, for example

```
public interface Consumer<T> {
    void accept(T t);
}
```

- The package `java.util.function` defines many new useful functional interfaces

Assigning lambda to a local variable

```
public interface Consumer<T> {
  void accept(T t);
}

void forEach(Consumer<Integer> action) {
  for (Integer i : items) {
    action.accept(i);
  }
}

List<Integer> intSeq = Arrays.asList(1,2,3);

Consumer<Integer> cnsmr = x -> System.out.println(x);
intSeq.forEach(cnsmr);
```

Functional interface used for **implementing** a lambda expression.

Method `forEach()` from list `List<Integer>` interface, which receives `Consumer<Integer>` as a parameter. `Consumer<Integer>` **implementation is generated** from the lambda expression.

Creating new `Consumer<Integer>` as a lambda expression.

Properties of the generated method

- The method generated from a Java lambda expression has the same signature as the method in the functional interface
- The type is the same as that of the functional interface to which the lambda expression is assigned
- The lambda expression becomes the body of the method in the interface

Variable capture

- Lambdas can interact with variables defined outside the body of the lambda
- Using these variables is called variable capture

Example: Local variable capture

```
public class LVCExample {
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1, 2, 3);

    int var = 10;

    intSeq.forEach(x -> System.out.println(x + var));
  }
}
```

- Local variables used inside the body of a lambda must be **final** or **effectively final**

Example: Static variable capture

```
public class SVCExample {

  private static int var = 10;

  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1, 2, 3);
    intSeq.forEach(x -> System.out.println(x + var));
  }
}
```


Method references (1)

- Method references can be used to pass an existing function in places where a lambda is expected
- The signature of the referenced method needs to match the signature of the functional interface method

Method references (2)

- Static method
  `ClassName::StaticMethodName`
  (e.g. `String::valueOf`)
- Constructor
  `ClassName::new`
  (e.g. `ArrayList::new`)
- Specific object instance
  `objectInstance::MethodName`
  (e.g. `myCooler::coolReactor`)
- Arbitrary object of a given type
  `ClassName::InstanceMethodName`
  (e.g. `Object::toString`)

Conciseness with method references

- We can rewrite the statement

```
intSeq.forEach(x -> System.out.println(x));
```

more concisely using a method reference

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

Stream API

- The new `java.util.stream` package provides utilities to support functional-style operations on streams of values
- A common way to obtain a stream is from a collection

```
Stream<T> stream = collection.stream();
```

- Streams can be sequential or parallel
- Streams are useful for selecting values and performing actions on the results

Stream operations

- An intermediate operation keeps a stream open for further operations
  - Intermediate operations are lazy
- A terminal operation must be the final operation on a stream
  - Once a terminal operation is invoked, the stream is consumed and is no longer usable

Exampe: Intermediate operations

`filter`

- Excludes all elements that don't match a predicate

`map`

- Performs a one-to-one transformation of elements using a function

A stream pipeline

- A stream pipeline has three components
  - A source such as a collection, an array, a generator function, or an IO channel
  - Zero or more intermediate operations
  - A terminal operation

Example: Filter elements, map them to numbers, and sum

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

- Here, widgets is a `Collection<Widget>`. We create a stream of `Widget` objects via `Collection.stream()`, filter it to produce a stream containing only the red widgets, and then transform it into a stream of int values representing the weight of each red widget. Then this stream is summed to produce a total weight

Example: Using lambdas and stream to sum the squares of the elements on a list

```
List<Integer> list = Arrays.asList(1, 2, 3);
int sum = list.stream()
              .map(x -> x * x)
              .reduce((x, y) -> x + y)
              .get();
System.out.println(sum);
```

- Here `map(x -> x * x)` squares each element and then `reduce((x, y) -> x + y)` reduces all elements into a single number