61FIT3JSD
Fall 2023

Lecture 12
Functional programming in Java

Lecture outline

- Introduction
- Functional programming
  - What is functional programming?
  - Why functional programming?
- Functional programming with Lambda expressions
- Streams

The imperative style

- Most of us are used to this style
- Source code tells the computer **what do to**

```java
public class FindNemo {
    public static void main(String[] args) {
        List<String> names =
            Arrays.asList("Dory", "Gill", "Bruce",
                "Nemo", "Darla", "Marlin", "Jacques");
        findNemo(names);
    }

    public static void findNemo(List<String> names) {
        // code omitted
    }
}
```

The imperative style

- ...as well as **how to do it**

```
public static void findNemo(List<String> names) {
    boolean found = false;
    for (String name : names) {
        if (name.equals("Nemo")) {
            found = true;
            break;
        }
    }
    if (found)
        System.out.println("Found Nemo!");
    else
        System.out.println("Sorry, Nemo not found!");
}
```

The declarative style

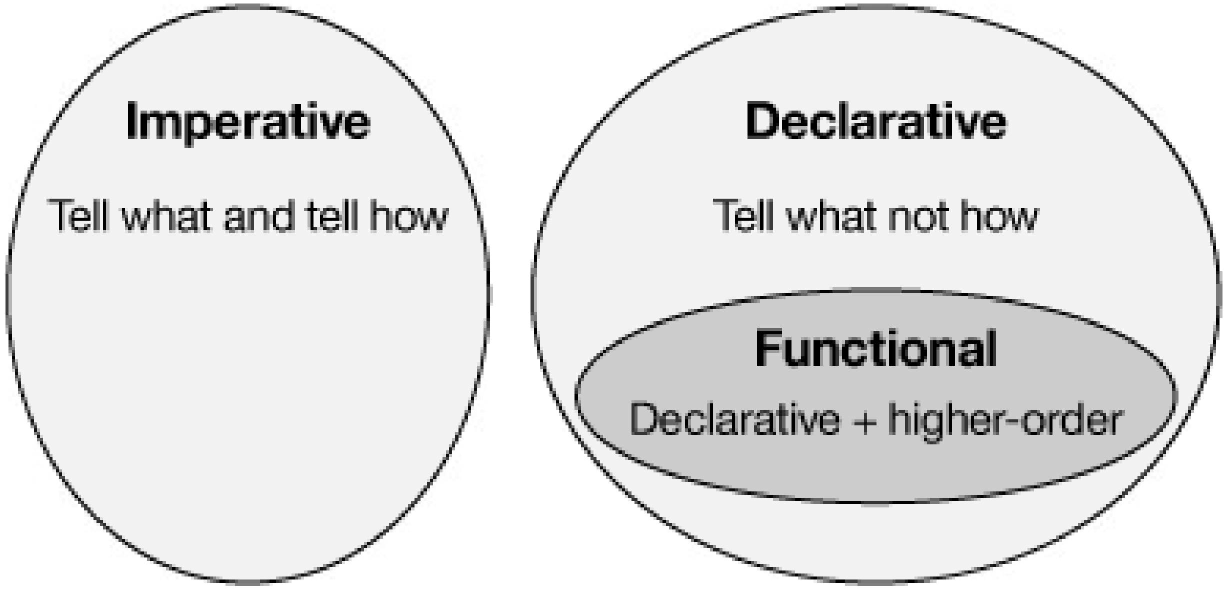
- Write **what to do**, but you leave the implementation details to the underlying library of functions

```
public static void findNemo(List<String> names) {
    if (names.contains("Nemo")) {
        System.out.println("Found Nemo");
    } else {
        System.out.println("Sorry, Nemo not found");
    }
}
```

✓ no garbage variables
✓ do not have to write loop

The functional style

- Combine declarative methods with higher order functions (HOF)
- HOF: a method or a function that can receive, create, or return a function



Functional programming

- What is functional programming?
  - Example
- Why functional programming?

What is functional programming?

- A style of programming
- Treats computations as the evaluation of mathematical functions
- Eliminates side effects
- Treats data as being immutable
- Expressions have referential transparency
- Functions can take functions as arguments and return functions as results *(HOF, method references)*
- Prefers recursion over explicit for-loops *(streams)*

Example: Map in imperative style

```java
public class UseMap {
    public static void main(String[] args) {
        Map<String, Integer> pageVisits = new HashMap<>();
        String page = "https://agiledeveloper.com";
        incrementPageVisit(pageVisits, page);
        incrementPageVisit(pageVisits, page);
        System.out.println(pageVisits.get(page));
    }

    public static void incrementPageVisit(
            Map<String, Integer> pageVisits, String page) {
        if (!pageVisits.containsKey(page)) {
            pageVisits.put(page, 0);
        }
        pageVisits.put(page, pageVisits.get(page) + 1);
    }
}
```

Example: map in the functional style

```java
public static void incrementPageVisit(
        Map<String, Integer> pageVisits, String page) {

    pageVisits.merge(page, 1,
            (oldValue, value) -> oldValue + value
    );

}
```

The `merge()` method:

- **1st argument:** the key whose value should be updated
- **2nd argument:** initial value if the key doesn't exist
- **3rd argument:** the remapping function
  (`oldValue`: existing value of the key, `value`: the 2nd argument)

Why 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

Functional programming with lambda expressions

- Lambda expressions
- Syntax
- Functional interfaces
- Variable capture
- Method references
- Default methods

Lambda expression

- Is a nameless function.
- Most important new addition in Java 8
- Related concepts: closures, anonymous functions, function literals

Benefits of Lambdas in Java 8

- 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 1:
Print a list of integers with a lambda

```
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 2:
A multiline lambda

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

intSeq.forEach(x -> {
    x += 2;
    System.out.println(x);
});
```

- Braces are needed to enclose a multi-line body in a Lambda expression.

Example 3:
A lambda with a 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 4:
A 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 8 Lambdas

- The Java 8 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 8 lambdas are assigned to functional interfaces.
- A functional interface is a Java interface with exactly one non-default method. E.g.

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

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

Assigning a Lambda to a local variable

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

// in List<T> class
void forEach(Consumer<T> action) {
    for (T item : items) {
        action.accept(item);
    }
}

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

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

Properties of the Generated Method

- The method generated from a Java 8 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

Local Variable Capture Example

```
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));
    }
}
```

Note: local variables used inside the body of a lambda must be `final` or effectively `final`

Static Variable Capture Example

```java
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

- 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

Summary of Method References

Method Reference Type	Syntax	Example
static	`ClassName::StaticMethodName`	`String::valueOf`
constructor	`ClassName::new`	`ArrayList::new`
specific object instance	`objectReference::MethodName`	`x::toString`
arbitrary object of a given type	`ClassName::InstanceMethodName`	`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);
```

Default Methods

Java 8 uses lambda expressions and default methods in conjunction with the Java collections framework to achieve backward compatibility with existing published interfaces

For a full discussion see Brian Goetz, Lambdas in Java: A peek under the hood.

https://www.youtube.com/watch?v=MLksirK9nnE

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.

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

Example 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:

1. A source such as a Collection, an array, a generator function, or an IO channel;
2. Zero or more intermediate operations; and
3. A terminal operation

Stream Example

```
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.

From Java Docs
Interface Stream<T>

Parting 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);
```

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

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

References

A lot of the material in this lecture is discussed in much more detail in these informative references:

The Java Tutorials, http://docs.oracle.com/javase/tutorial/java/index.html

Lambda Expressions, http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Adib Saikali, Java 8 Lambda Expressions and Streams, www.youtube.com/watch?v=8pDm_kH4YKY

Brian Goetz, Lambdas in Java: A peek under the hood. www.youtube.com/watch?v=MLksirK9nnE

Venkat Subramaniam, Java 8 Idioms