Functional Programming in Java

INT103 Advanced Programming

2020-2

Asst. Prof. Kriengkrai Porkaew, PhD.

A Computer Program

A computer program is **a collection of instructions** that performs a specific task when executed by a computer.

A computer program is usually written in a human-readable form (called **source code**) by a computer programmer in **a programming language**.

A programming language is a formal language, which comprises a set of instructions that produce various kinds of output.

https://en.wikipedia.org/wiki/Computer_program
https://en.wikipedia.org/wiki/Programming_language

PROGRAMMING LANGUAGES

• Imperative Programming Languages

- **Imperative programming** is **a programming paradigm** that uses statements that change a program's state. An imperative program consists of commands for the computer to perform.
- **Structured programming** is **a programming paradigm** aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines.
- **Procedural programming** is **a programming paradigm**, derived from structured programming, based upon the concept of the *procedure call*. Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out.

https://en.wikipedia.org/wiki/Imperative_programming
https://en.wikipedia.org/wiki/Structured_programming
https://en.wikipedia.org/wiki/Procedural_programming

OBJECT-ORIENTED PROGRAMMING

- Imperative > Structured > Procedural programming
  - A program can be viewed as a collection of **data** that are stored in variables and **functions** that consist of statements that change the data that are stored in variables.
- **Object-Oriented Programming**
  - Object-oriented programming is derived from imperative > structured > procedural programming.
  - data structures and functions on that data structures are combined to form a small unit of program called **object**.
  - Objects that share the same data structures and functions are objects of the same type called **class**.
  - Functions on objects are called **methods**.
    Data structures in objects are called **attributes** or **fields**.
  - **Objects are stored in variables**.
  - In a sense, **OOP focuses more on data** while **functions are just parts of data**.
    Data can be changed dynamically but functions cannot be changed.

DECLARATIVE APPROACH VS. IMPERATIVE APPROACH

- Imperative > Structured > Procedural programming
  - A program can be viewed as a collection of **data** that are stored in variables and **functions** that consist of statements that change the data that are stored in variables.
- **Functional Programming**
  - Functional programming languages are **declarative**:
    describing what is needed to be done (declarative); not how to do it (imperative).
  - **Functions are first-class citizen**
    - Functions can be **treated as values**, just like data.
    - Functions can be **stored in variables**, just like data.
    - Functions can be **passed as arguments** to other functions, just like data.
    - Functions can be **returned as results** from other functions, just like data.
    - Data have types and **functions also have types**.
  - A function may not need to have a name if it is defined and assigned to a variable directly. This function is an **anonymous function**. The expression that defines the function is called **lambda expression**.
  - Functions that can receive other functions as parameters or return other functions as results are called **higher-order functions**.

A little bit of History of Functional Programming Languages

- **Lambda Calculus** (1930s – by **Alonzo Church**)
  - Turing complete
- **Lisp** (1958 – by **John McCarthy**)
  - Second oldest high-level programming language (after FORTRAN - 1957)
- **Scheme** (1975)
  - Lisp family, lexical scope, tail-call optimization, continuation
- **Haskell** (1990 – named after **Haskell Curry**)
  - Purely functional programming language
- **Clojure** (2007)
  - Lisp family

WHY FUNCTIONAL PROGRAMMING IN JAVA?

- Functional Programming Concepts
  - Functional programming languages are **declarative**:
    describing what is needed to be done; not how to do it.
  - Functional programming prefers **immutable**, **no state**, **and** **no side-effect**:
    data should be immutable; functions should not access any state outside the functions;
    and executions of functions should have no side-effect to external states;
    functions do not change their behaviors: If same input then same output.
- Object-oriented programming concepts
  - OOP, in a sense, are quite the opposite of functional programming concepts.
  - Most objects are mutable and full of states (states = data in the attributes).
  - Methods cause lots of side-effect (side-effect = change data in the attributes).
- Why Functional Programming in Java?
  - More readable code, more concise code vs. boilerplate code
  - Easier to go parallel with functional programming:
    "what to do – not how to do", "immutable", "side-effect free",

Java: Class -> Anonymous Class -> Lambda Expression


```java
interface Greeting {
    public void greet();
}
```

() -> void

Take no argument and return nothing

```java
class Hello implements Greeting {
    @Override
    public void greet() {
        System.out.println("Hello Class.");
    }
}
```

```java
public class FunctionalTest {
    public static void main(String[] args) {
        Greeting [] array = new Greeting[3];
        array[0] = helloClass();
        array[1] = helloAnonymous();
        array[2] = helloLambda();
        for (Greeting var : array) {
            var.greet();
        }
    }
```

function invocation

```java
private static Greeting helloClass() {
    return new Hello();
}
```

```java
private static Greeting helloAnonymous() {
    return new Greeting() {
        @Override
        public void greet() {
            System.out.println("Hello Anonymous.");
        }
    };
}
```

anonymous class

Output =>

```
Hello Class.
Hello Anonymous.
Hello Lambda.
```

```java
private static Greeting helloLambda() {
    return () -> System.out.println("Hello Lambda.");
}
```

Lambda Expression => anonymous function

concise code

Syntax of Lambda Expression

- Lambda Expression =
  - (*LIST_OF_VARIABLES*) - > { *LIST_OF_STATEMENTS_ENDED_WITH_RETURN* }
  - *LIST_OF_VARIABLES* = *VARIABLE* [, *VARIABLE*] *
  - *VARIABLE* = **DATA_TYPE VARIABLE_NAME**
  - **DATA_TYPE** can be omitted.
  - If there is ***only one variable*** in the ***LIST_OF_VARIABLES***, ( ) can be omitted.
  - If there is ***only one statement*** in the ***LIST_OF_STATEMENTS***, { } can be omitted.
  - If { } is omitted, the keyword **return** in the statement can be omitted too.
- **Lambda Expression** is an expression, so it can be …
  - assigned to a variable (waiting to be executed),
  - stored in an array or a collection as a value (waiting to be executed),
  - returned from a function as a value (waiting to be executed),
  - passed to another function as an argument (waiting to be executed).

  It can be executed using the single abstract method in the functional interface.

Syntax of Lambda Expression

```
String substring(String str, int offet, int length) {
    if (offet<0 || length<=0 || offset+length>str.length())
        throw new IllegalArgumentException();
    return str.substring(offset, offset+length);
}
```

From a method
to a lambda expression

```
(String str, int offet, int length) -> {
    if (offet<0 || length<=0 || offset+length>str.length())
        throw new IllegalArgumentException();
    return str.substring(offset, offset+length);
}
```

If data type is omitted.

```
(str, offet, length) -> {
    if (offet<0 || length<=0 || offset+length>str.length())
        throw new IllegalArgumentException();
    return s.substring(offset, offset+length);
}
```

If arguments are not validated.

```
(str, off, len) -> {
    return s.substring(off, off+len);
}
```

If there is only one statement,
{ } and return can be omitted.

```
(str, off, len) -> str.substring(off, off+len);
```

Examples of Lambda Expressions

Since functions/lambdas also need to have a type:
Interfaces act as the types of functions/lambdas.
These interfaces must have **only one abstract method**.
These interfaces are called **functional interface**.

```
public class FunctionalTest2 {
    public static void main(String[] args) {
        Greeting1 g1;
        Greeting2 g2;
        Greeting3 g3, g4;
        g1 = s -> System.out.println("G1: Hello, " + s);
        g2 = (s, m) -> System.out.println("G2: " + m + ", " + s);
        g3 = s -> "G3: Hello, " + s;
        g4 = s -> {
            String cap = s.toUpperCase();
            return "G4: Hello, " + cap;
        };

        g1.greet("you");
        g2.greet("Lambda", "Good Day");
        System.out.println(g3.greet("Simple Lambda"));
        System.out.println(g4.greet("A Little Complex Lambda"));
    }
}
```

Lambda Expressions

```
interface Greeting1 {
    public void greet(String someone);
}
```

String -> void

```
interface Greeting2 {
    public void greet(String someone, String message);
}
```

(String, String) -> void

```
interface Greeting3 {
    public String greet(String someone);
}
```

String -> String

Output =>

```
G1: Hello, you
G2: Good Day, Lambda
G3: Hello, Simple Lambda
G4: Hello, A LITTLE COMPLEX LAMBDA
```

Another Example of Lambda Expression in Java

```java
public class OOCalculator implements Calculator {
    double left, right;
    String operator;

    public OOCalculator(double left, double right, String symbol) {
        this.left = left; this.right = right;
        this.operator = operation(symbol);
    }

    private String operation(String symbol) {
        switch (symbol) {
            case "+": case "*": case "-": case "/":
            case "%": return symbol;
            default: throw new IllegalArgumentException();
        }
    }

    @Override
    public double compute() {
        switch (operator) {
            case "+": return left + right;
            case "*": return left * right;
            case "-": return left - right;
            case "/": return left / right;
            case "%": return left % right;
            default: throw new IllegalStateException();
        }
    }
}
```

```java
@FunctionalInterface
public interface DoubleBinaryOperator {
    double applyAsDouble(double left, double right);
}
```

(double, double) -> double

```java
public class FunctionalCalculator implements Calculator {
    double left, right;
    DoubleBinaryOperator operator;

    public FunctionalCalculator(double left, double right,
            String symbol) {
        this.left = left; this.right = right;
        this.operator = operation(symbol);
    }

    private DoubleBinaryOperator operation(String symbol) {
        switch (symbol) {
            case "+": return (d1, d2) -> d1 + d2;
            case "-": return (d1, d2) -> d1 - d2;
            case "*": return (d1, d2) -> d1 * d2;
            case "/": return (d1, d2) -> d1 / d2;
            case "%": return (d1, d2) -> d1 % d2;
            default: throw new IllegalArgumentException();
        }
    }

    @Override
    public double compute() {
        return operator.applyAsDouble(left, right);
    }
}
```

A function is stored in a variable

Functions (lambda expressions) are returned as results

The function is executed (the Java way)

Functional Programming in Java

- Java is a **strongly typed** programming language
  - Since functions can be treated as values, **functions need to have types**.
  - Types of functions, in a sense, are **function signatures**.
  - E.g.,
    - a function that receives no argument and returns a boolean-value result have the following type: **() -> boolean** (called **BooleanSupplier**)
    - A function that receives two double-value arguments and returns a double-value result have the following type: **(double, double) -> double** (called **DoubleBinaryOperator**)
  - Package: **java.util.function**
    - Provides 40+ types of functions in the form of interface; each of which is annotated with **@FunctionalInterface**
    - Interfaces that are annotated with @FunctionalInterface are interfaces that **can have only one abstract, non-static, non-default method** and the method signature is the function signature.
    - 40+ types of functions can be grouped into 3 types:
      - **Suppliers**: **receive nothing** as an input but **return** an output value.
      - **Consumers**: **receive** one or two input arguments but **return nothing** as output.
      - **Functions**: **receive** one or two input arguments and **return** an output value.
    - Developers may create their own types of functions anytime, just like creating their own classes.

43 @FunctionalInterface in **java.util.function** package

No Input Argument

Supplier : get()

Supplier<T> : () -> T
BooleanSupplier : () -> boolean
IntSupplier : () -> int
LongSupplier : () -> long
DoubleSupplier : () -> double

One Input Argument (**Unary**)

Function: Predicate, Operator, Function

(Bi)Function : apply()

Function<T,R> : T -> R
IntFunction<R> : int -> R
LongFunction<R> : long -> R
DoubleFunction<R> : double -> R
ToIntFunction<T> : T -> int
ToLongFunction<T> : T -> long
ToDoubleFunction<T> : T -> double
IntToLongFunction : int -> long
IntToDoubleFunction : int -> double
LongToIntFunction : long -> int
LongToDoubleFunction : long -> double
DoubleToIntFunction : double -> int
DoubleToLongFunction : double -> long

andThen(…)

and(), or(…),
negate(), isEqual(…)

No Returned Value
(Bi)Consumer : accept()

Consumer<T> : T -> void
IntConsumer : int -> void
LongConsumer : long -> void
DoubleConsumer : double -> void

Return boolean
(Bi)Predicate : test()

Predicate<T> : T -> boolean
IntPredicate : int -> boolean
LongPredicate : long -> boolean
DoublePredicate : double -> boolean

Input/Output : Same Type
(Unary/Binary)Operator

UnaryOperator<T> : T -> T
IntUnaryOperator : int -> int
LongUnaryOperator : long -> long
DoubleUnaryOperator : double -> double

Two Input Arguments (**Binary**)

BiConsumer<T,U> : T, U -> void
ObjIntConsumer<T> : T, int -> void
ObjLongConsumer<T> : T, long -> void
ObjDoubleConsumer<T> : T, double -> void

BiPredicate<T,U> : T, U -> boolean

BinaryOperator<T> : T, T -> T
IntBinaryOperator : int, int -> int
LongBinaryOperator : long, long -> long
DoubleBinaryOperator : double, double -> double

BiFunction<T,U,R> : T, U -> R
ToIntBiFunction<T,U> : T, U -> int
ToLongBiFunction<T,U> : T, U -> long
ToDoubleBiFunction<T,U> : T, U -> double

Other @FunctionalInterface elsewhere; e.g.,

Runnable : run() : () -> void

Callable<V> : call() : () -> V

Comparator<T> : compare() : T, T -> int

Java Stream API

Implementing Higher-Order Functions

- filter()
- map()
- reduce()
- forEach()
- …
- count()
- min()
- max()
- …

Hierarchy For Package java.util.stream

Package Hierarchies:
All Packages

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Class Hierarchy

- java.lang.**Object**
  - java.util.stream.**Collectors**
  - java.util.stream.**StreamSupport**

Interface Hierarchy

- java.lang.**AutoCloseable**
  - java.util.stream.**BaseStream**<T,S>
    - java.util.stream.**DoubleStream**
    - java.util.stream.**IntStream**
    - java.util.stream.**LongStream**
    - java.util.stream.**Stream**<T>
- java.util.stream.**Collector**<T,A,R>
- java.util.function.**Consumer**<T>
  - java.util.stream.**Stream.Builder**<T>
- java.util.function.**DoubleConsumer**
  - java.util.stream.**DoubleStream.Builder**
- java.util.function.**IntConsumer**
  - java.util.stream.**IntStream.Builder**
- java.util.function.**LongConsumer**
  - java.util.stream.**LongStream.Builder**