Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

SUPSI

Lambda Expressions

Object Oriented Programming

Tiziano Leidi

09.11.2021

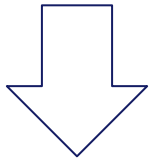
Lambda Expressions

Anonymous classes can be effective. However, if the contained code is simple, the scaffolding code required for the implementation easily becomes an overkill and limits the exploitation potential.

As an alternative more compact version, Java features lambda expressions.

Anonymous class:

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Detected by Anonymous
Class");
    }
});
```



Lambda expression:

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(e -> System.out.println("Detected by
Lambda"));
```

Lambda Expressions

Lambda expressions allow to manage functions and procedures (which contain lines of code) in a similar way to data.

For example, lambdas can be exploited to provide functionality (a block of code to execute) through a parameter of a method, or to return functionality through the return value of a method.

```
class Calculator {
    private Operation[] operations = new Operation[2];
    private int cnt = 0;

    public Calculator() {
        addOperation((a, b) -> a + b);
        addOperation((double a, double b) -> {
            return a - b;
        });
    }

    private void addOperation(Operation operation) {
        operations[cnt++] = operation;
    }

    public void showResults(double a, double b) {
        for (Operation operation : operations)
            System.out.println("Result: " + operation.execute(a, b));
    }
}
```

```
interface Operation {
    double execute(double a, double b);
}
```

```
public class CalculatorTest {
    public static void main(String[] args) {
        new Calculator(). showResults(42.42, 666.666);
    }
}
```

Functional Interface

Each lambda expression is represented by an interface containing only a single abstract method, and default methods (if needed).

This type of interface is called Functional Interface and is annotated with @FunctionalInterface.

```
@FunctionalInterface
interface Operation {
    double execute(double a, double b);
}
```

Syntax of Lambda Expressions

The syntax for lambda expressions is as follows:

- the list of parameters of the functional interface method is followed by an arrow (->), followed by the method's implementation code.

```
(int x, int y) -> x + y
() -> 42
(String s) -> { System.out.println(s); }
```

- The implementation can be either a single line of code or a block of code.

Syntax of Lambda Expressions

- There is no need to declare the type of the parameters:

```
(int a, int b) -> a + b
(a, b) -> a + b
```

- In case of a single parameter, round brackets are not required. In case of multiple parameters, brackets are instead mandatory:

```
a -> 2 * a
(a, b) -> a * b
```

Syntax of Lambda Expressions

- If the body is composed of a single instruction, curly brackets are optional:

```
a -> { System.out.println("a: " + a); }
a -> System.out.println("a: " + a)
```

- If the body has a single expression that returns a value, the `return` keyword is not mandatory:

```
(a, b) -> {return a + b;}
(a, b) -> a + b
```

Method References

In some situations, a lambda expression may just call a method. In these cases, it is often more convenient to directly specify a reference to method.

Method references are compact lambda expressions that directly reference existing methods.

- Static method:

  Class::staticMethod

  equivalent to: *(args) -> Class.staticMethod(args)*
- Instance method of an object

  obj::instanceMethod

  equivalent to: *(args) -> obj.instanceMethod(args)*
- Instance method of an object type:

  Class::instanceMethod

  equivalent to: *(args) -> obj.instanceMethod(args)* for given obj
- Constructor

  Class::new

  equivalent to: *(args) -> new ClassName(args)*

Examples

```
word -> StringUtils.capitalize(word)
StringUtils::capitalize

(a, b) -> bikeFrameSizeComparator.compare(a, b)
bikeFrameSizeComparator::compare

(a, b) -> a.compareTo(b)
Integer::compareTo
```

… it also depends on the context of use (in particular the arguments available).

```
class Calculator {
    private static class MathUtilities {
        private static double multiply(double a, double b) {
            return a * b;
        }

        private double divide(double a, double b) {
            return a / b;
        }
    }

    private Operation[] operations = new Operation[4];
    private int cnt = 0;

    public Calculator() {
        addOperation((a, b) -> a + b);
        addOperation((double a, double b) -> {
            return a - b;
        });
        addOperation(MathUtilities::multiply);
        addOperation(new MathUtilities()::divide);
    }

// ...
```

```
interface Operation {
    double execute(double a, double b);
}
```

```
// ...

    private void addOperation(Operation operation) {
        operations[cnt++] = operation;
    }

    public void showResults(double a, double b) {
        for (Operation operation : operations)
            System.out.println("Result: " + operation.execute(a, b));
        }
    }
}
```

```
public class CalculatorTest {
    public static void main(String[] args) {
        new Calculator(). showResults(42.42, 666.666);
    }
}
```

Closures

A closure is a special kind of object that combines two things: a function, and the environment in which that function was created.

The environment consists of any local variables that were in-scope at the time that the closure was created.

Closures are mandatory: the created function is allowed to exist, even after the local context in which it was created has completed its execution.

```java
@FunctionalInterface
interface NumToMonth {
    public String convertToMonth(int x);
}

public class ClosuresExample {
    public static NumToMonth getNumToMonthConverter() {
        String[] months = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
                "Aug", "Sep", "Oct", "Nov", "Dec"};
        return n -> (n > 0 && n <= months.length) ? months[n - 1] : null;
    }

    public static void main(String[] args) {
        System.out.println(getNumToMonthConverter().convertToMonth(8));
    }
}
```

Closures

There are programming languages (such as JavaScript) that allow a closure to modify the values of the captured local variables of the environment.

Such solutions provide more flexibility, but allows the possibility of introducing side effects, which could lead to unwanted errors.

Lambda functions that modify free variables are considered a bad practice.

Closures in Java

Instead of maintaining a pointer to the enclosing scope (like in JavaScript), Java only saves the value of the free variables to let them be used inside lambda expressions or anonymous classes.

For this reason, the type of variables that can be accessed from lambda expressions and anonymous classes is limited to only final and effectively final ones.

Closures in Java

However, in case of objects, the "final" keyword only avoids modification of the reference of the object. The contents of the objects (on the heap) is still subject to modifications.

Therefore, side effects are also possible in Java and needs to be prevented (in particular in multi-threaded programs).

```java
class ValueHolder {
    int value = 0;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}

class ValueConsumer {
    public void execute(Supplier<ValueHolder> supplier) {
        ValueHolder holder = supplier.get();
        System.out.println("Current value: " + holder.getValue());

        // ...

        holder.setValue(50);
    }
}
```

```java
public class SideEffectExample {
    public static void main(String[] args) {
        final ValueHolder valueHolder = new ValueHolder();
        valueHolder.setValue(20);
        final ValueConsumer valueConsumer = new ValueConsumer();
        valueConsumer.execute(() -> valueHolder);

        // ...

        System.out.println("Final value: " + valueHolder.getValue());
    }
}
```

Standard Functional Interfaces

Java provides a multitude of standard interfaces that can be used as functional interfaces.

These standard functional interfaces are frequently used in the Java library classes.

Example from the Javadoc:

`boolean`	**`removeIf`**`(`**`Predicate`**`<? super E> filter)` Removes all of the elements of this collection that satisfy the given predicate.
`protected void`	**`removeRange`**`(int fromIndex, int toIndex)` Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive.
`void`	**`replaceAll`**`(`**`UnaryOperator`**`<E> operator)` Replaces each element of this list with the result of applying the operator to that element.

Standard Functional Interfaces

Common standard functional interfaces are:

Interface name	Arguments	Returns	Example
`Predicate<T>`	`T`	`boolean`	Has this album been released yet?
`Consumer<T>`	`T`	`void`	Printing out a value
`Function<T,R>`	`T`	`R`	Get the name from an `Artist` object
`Supplier<T>`	None	`T`	A factory method
`UnaryOperator<T>`	`T`	`T`	Logical not (`!`)
`BinaryOperator<T>`	`(T, T)`	`T`	Multiplying two numbers (`*`)

Function and BiFunction

A Function takes an argument (object of type T) and returns an object (object of type R):

```
@FunctionalInterface
public interface Function<T, R> {
   R apply(T t);
}
```

A BiFunction takes two arguments and returns an object:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
   R apply(T t, U u);
}
```

```
Function<String, Integer> func = x -> x.length();
System.out.println(func.apply("Example"));


// takes two Integers and return an Integer
BiFunction<Integer, Integer, Integer> func1 = (x1, x2) -> x1 + x2;
System.out.println(func1.apply(2, 3));

// take two Integers and return an Double
BiFunction<Integer, Integer, Double> func2 = (x1, x2) -> Math.pow(x1, x2);
System.out.println(func2.apply(2, 4));

// take two Integers and return a List<Integer>
BiFunction<Integer, Integer, List<Integer>> func3 = (x1, x2) -> Arrays.asList(x1 + x2);
System.out.println(func3.apply(2, 3));
```

Function and BiFunction Methods

Function methods:

`andThen(Function<? super R,? extends V> after)`
Returns a composed function that first applies this function to its input, and then applies the `after` function to the result.

`apply(T t)`
Applies this function to the given argument.

`compose(Function<? super V,? extends T> before)`
Returns a composed function that first applies the `before` function to its input, and then applies this function to the result.

`identity()`
Returns a function that always returns its input argument.

BiFunction methods:

`andThen(Function<? super R,? extends V> after)`
Returns a composed function that first applies this function to its input, and then applies the `after` function to the result.

`apply(T t, U u)`
Applies this function to the given arguments.

Predicate

A Predicate accepts an argument and returns a boolean:

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

Usually, it is used to apply a filter to a collection of objects.

Predicate Methods

Predicate methods:

`and(Predicate<? super T> other)`
Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.

`isEqual(Object targetRef)`
Returns a predicate that tests if two arguments are equal according to **`Objects.equals(Object, Object)`**.

`negate()`
Returns a predicate that represents the logical negation of this predicate.

`or(Predicate<? super T> other)`
Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.

`test(T t)`
Evaluates this predicate on the given argument.

BiPredicate

A BiPredicate accepts two arguments and returns a boolean:

```
@FunctionalInterface
public interface BiPredicate<T, U> {
   boolean test(T t, U u);
}
```

The behaviour is identical to Predicate.

BiPredicate Methods

BiPredicate methods:

and(**BiPredicate**<? super **T**,? super **U**> other)
Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.

negate()
Returns a predicate that represents the logical negation of this predicate.

or(**BiPredicate**<? super **T**,? super **U**> other)
Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.

test(**T** t, **U** u)
Evaluates this predicate on the given arguments.

UnaryOperator and BinaryOperator

UnaryOperator is a functional interface that extends Function.
The UnaryOperator takes one argument, and returns a result of the same type of its arguments.

Similarly, BinaryOperator is a functional interface that extends BiFunction.
The BinaryOperator takes two arguments of the same type and returns a result of the same type of its arguments.

```
Function<Integer, Integer> f1 = x -> x * 2;
System.out.println(f1.apply(2));

UnaryOperator<Integer> f2 = x -> x * 2;
System.out.println(f2.apply(2));

BiFunction<Integer, Integer, Integer> f3 = (x1, x2) -> x1 + x2;
System.out.println(f3.apply(2, 3));

BinaryOperator<Integer> f4 = (x1, x2) -> x1 + x2;
System.out.println(f4.apply(2, 3));
```

Supplier

A Supplier takes no arguments and returns a result:

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

```
Supplier<LocalDateTime> s = () -> LocalDateTime.now();
LocalDateTime time1 = s.get();
System.out.println(time1);

DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
Supplier<String> s1 = () -> dtf.format(LocalDateTime.now());
String time2 = s1.get();
System.out.println(time2);
```

Summary

- From anonymous classes to lambda expressions
- Details about the syntax of lambda expressions
- Introduction to functional interfaces
- Method references
- Details about Java closures
- Standard functional interfaces in Java