CONCEPTS OF FUNCTIONAL PROGRAMMING

Yannick Chartois / @ychartois

BODIL STOKKE

What Every Hipster Should Know About Functional Programming

Vimeo / Parleys

MY FIRST QUESTION

Now that we have lambdas in Java 8, can we use FP concepts?

EXTENDED QUESTION

Is there an advantage to use these concepts in other language with First Class Function as Javascript

1. FIRST CLASS FUNCTION

Definition

“A programming language is said to have first-class functions if it treats functions as first-class citizens”

1. FIRST CLASS FUNCTION

Code:

```
Function< String, String > hello = (String s) -> "hello " + s;
```

Result:

```
hello ;
// BaseConcepts$$Lambda$1@a09ee92
// != BaseConcepts@30f39991

hello.apply("Erouan") ;
// hello Erouan
```

2. HIGH ORDER FUNCTION

Definition

“It's a function that takes one or more function as parameters or that returns a function”

2. HIGH ORDER FUNCTION

Code:

```
Function< String, String > twice( Function< String, String > f ) {
    return (String s) -> f.apply( f.apply(s) );
}
```

Resultat:

```
twice(hello).apply("Erouan");
// hello hello Erouan
```

2. HIGH ORDER FUNCTION

Code:

```
hello = (s) -> return "Hello " + s

twice = (func, s) ->
    return func func s
```

Resultat:

```
twice(hello,"Erouan")
// hello hello Erouan
```

3. CURRYING

Definition

"Currying is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument"

3. CURRYING

Previous code:

```
twice = (func, s) ->
    return func func s
```

Currying code:

```
twice = ( func ) -> ( s ) ->
    return func func s
```

Result:

```
twice(hello)("Erouan")
// hello hello Erouan
```

4. FUNCTOR

Definition

“A functor is a collection of X that can apply a function $f : X \longrightarrow Y$ over itself to create a collection of Y.”

4. FUNCTOR - MAP

Code:

```
List< String > map( Function< String, String > f
                        , List< String > values ) {
    List< String > toReturn = new ArrayList< >();
    for( String current : values ) {
        toReturn.add( f.apply(current) );
    }
    return toReturn;
}
```

Result:

```
List< String > confs =
                Arrays.asList( new String[]{"Corkdev", "devoxx", "javac

map( s -> s.toUpperCase(), confs );

// [CORKDEV, DEVOXX, JAVAONE]
```

With Java 8:

```
confs.stream().map( s -> s.toUpperCase() ).collect( Collectors.toList()
```

With Coffescript:

```
["Corkdev", "devoxx", "javaone"].map (el) -> el.toUpperCase()
```

4. FUNCTOR - FILTER

Code:

```
List< String > filter( Function< String, Boolean > f,
                                List< String > values ) {
    List< String > toReturn = new ArrayList< >();
    for( String current : values ) {
        if ( f.apply(current) )
            toReturn.add( current );
    }
    return toReturn;
}
```

Result:

```
List< String > confs =
             Arrays.asList( new String[]{"jug", "devoxx", "javaone"} );

filter(s -> s.contains("j"), confs) ;

// [jug, javaone]
```

With Java 8:

```
confs.stream().filter( s -> s.contains("j") ).collect(Collectors.toList(
```

```
coms.stream().filter(s -> s.contains("j")).collect(Collectors.toList()
```

With Coffescript:

```
["Corkdev", "devoxx", "javaone"].filter (el) -> el.indexOf('k') != -1
```

5. REDUCE / FOLD

Definition

“Fold is a family of higher order functions that process a data structure in some order and build a return value”

5. REDUCE / FOLD

Code:

```
String reduce( BinaryOperator< String > op , List< String > values ) {
    String toReturn = "";
    for( String current : values ) {
        toReturn = toReturn.isEmpty() ? current : op.apply(toReturn, cur
    }
return toReturn; }
```

Result:

```
List< String > confs = Arrays.asList( "Corkdev", "devoxx", "javaone" );
reduce( (s1, s2) -> s1 + ", " + S2, confs );
// Corkdev, devoxx, javaone
```

With Java 8:

```
confs.stream().reduce((s1, s2) -> s1 + ", " + s2 ).get() )
```

With Coffescript:

```
["Corkdev", "devoxx", "javaone"].reduce (e1, e2) -> e1 + ", " + e2
```

6. COMBINATOR

Definition

“One definition of a combinator is a function with no free variables.”

6. COMBINATOR - NULL COMBINATOR

Constat:

```
List< String > confs2 = Arrays.asList( new String[]
                        {"jug", "devoxx", "javaone", null} );

map( s -> s.toUpperCase(), confs2);
// Exception in thread "main" java.lang.NullPointerException
```

Code:

```
Function< String, String > nullCheck( Function< String, String > f ) {
    return (String s) -> s == null ? "null" : f.apply(s);
}
```

Result:

```
map( nullCheck(s -> s.toUpperCase()), confs2)
// [JUG, DEVOXX, JAVAONE, null]
```

7. COMPOSITION

Definition

“Combine several functions to create a new function”

7. COMPOSITION

Code:

```
Function< String, String > compose ( Function< String, String > f1,
                                     Function< String, String > f2  )
    return (String s) -> f1.apply( f2.apply(s) );
}
```

Result:

```
Function< String, String > up = (String s) -> s.toUpperCase();
Function< String, String > hello = (String s) -> "hello " + s;

up.apply( hello.apply("Erouan") );

compose( up, hello).apply("Erouan") ;
// HELLO EROUAN
```

With Java 8:

```
hello.andThen(up).apply("Erouan")
up.compose(hello).apply("Erouan")
```

7. COMPOSITION - JAVASCRIPT

Code Coffee:

```
compose = (fs...) -> fs.reduce (f, g) -> (as...) -> f g as...
```

Code JS:

```
compose = function() {
    var fs;
    fs = 1 <= arguments.length ? slice.call(arguments, 0) : [];
    return fs.reduce(function(f, g) {
        return function() {
            var as;
            as = 1 <= arguments.length ? slice.call(arguments, 0) : [];
            return f(g.apply(null, as));
        };
    });
};
```

Result:

```
up = (s) -> s.toUpperCase()
hello = (s) -> "Hello " + s

compose(up, hello) "Pierre"
// HELLO PIERRE
```

WHY??

```
names = ["Pierre", "John", "Colm", "Petra", "Lenka"]
up = (s) -> s.toUpperCase()
hello = (s) -> "Hello " + s
```

Because for me that:

```
names.filter( (el) -> el.indexOf('k') == -1 )
    .map( (el) -> compose(up, hello)(el) )
    .reduce( (e1, e2) -> e1 + ", " + e2 )
```

Is more readable than:

```
acc = ""
for name in names
    if name.indexOf('k') == -1
        if acc != ""
            acc += ", " + up hello name
        else
            acc += up hello name
```

END !

SOURCES: PROGFONCJAVA8

TWITTER: @YCHARTOIS