▶

note 4 views

Returning more than one result from a method

While looking at a couple of students' Johnniac labs, I noted that they had written methods that needed to return more than one value. Neither Grace nor most popular languages allow this — which is odd, really, because all languages allow more than one argument.

So I though that I would write down here some ways to get around this problem. As motivation, consider this method

```
method parse(instruction:String) {
    // split instruction into opCode and operand
    def opCode = instruction.subStringFrom 1 size 2
    def operand = instruction.subStringFrom 3
    return ...
}
```

What should it return? There are several options.

Option 0: Use a global variable

You may be tempted to have two global variables, `opCode` and `operand`, and assign to them instead of returning a result. This way lies madness! You have to remember that this method writes to global variables, and that other methods that use these values read from them, and that no other methods should do so. The code that requests these methods becomes unreadable because the reader can't see where the values are flowing. **Don't do this.**

Option 1: Return a Sequence

Perhaps the most obvious solution is to return a sequence containing the two results:

```
method parse(instruction:String) {
    // split instruction into opCode and operand
    def opCode = instruction.subStringFrom 1 size 2
    def operand = instruction.subStringFrom 3
    return sequence.with(opCode, operand)
}
```

Then the client code has to get the results out of the sequence:

```
        answer := parse(inputWord)
        def operation = answer.first
        def argument = answer.second
        // code that manipulates operation and argument
```

This isn't very convincing in this example, because the code to extract the results is as long as the parse method, and a lot harder to understand. It would be better to just write

```
def operation = instruction.subStringFrom 1 size 2
```

Option 2: Return an Object

Readability is improved if, instead of a sequence, we create a custom object containing the two results:

```
method parse(instruction:String) {
    // split instruction into opCode and operand
    def result = object {
        def opCode is public = instruction.subStringFrom 1 size 2
        def operand is public = instruction.subStringFrom 3
    }
    return result
}
```

The client code could get the results out of the object:

```
        answer := parse(inputWord)
        def operation = answer.opCode
        def argument = answer.operand
        // code that manipulates operation and argument
```

but in fact **defs** are unnecessary — the client code can instead use `answer.opcode` and `answer.operand` directly.

Option 3: Un-Ask the Question

The question that we started out with was how to return more than one result from a method. Let's un-ask that question: why would you want to? In the given example, it's much simpler to write two separate methods:

```
method opCode(instruction:String) {
    // extract the opCode from instruction
    instruction.subStringFrom 1 size 2
}

method operand(instruction:String) {
    // extract the operand from instruction
    instruction.subStringFrom 3
}
```

This is a good solution quite a lot of the time. It's not such good solution when you have to do quite lot of work to compute the first result, but then the second one is almost free. For example, splitting a filename like "homework.grace" into the base ("homework") and the extension (".grace") requires searching for the dot. Once you have found it, and extracted the base, it seems like a shame to repeat that work in order to find the extension. But even in this case, it's really not so bad, and the extra work of searching for the dot twice may well be less than creating an object to contain the result.

Option 4: Create an Object with Behaviour

Once you have seen Option 2 — Return an Object — you might start to think about what that new object is actually representing. It's an instruction, right? So lets's give it enough behaviour so that we can *use instructions instead of strings* throughout the whole program.

```
method instructionfromString(instruction:String) {
    // create an object that represents an instruction
    object {
        def opCode is public = instruction.subStringFrom 1 size 2
            // the opcode of this instruction
        def operand is public = instruction.subStringFrom 3
            // the operand of this instruction
        def asString is public = opCode ++ operand
    }
}
```

This is not really any different from the code in Option 2, except that now we are thinking of `fromString` as a factory method, and our program will use the objects that it returns in the place of strings. (Indeed, this can, and probably should, be written with the **factory method** syntax).

Now that we have instructions, you might decide that it makes sense to put even more behavior into the instructions, for example, they might be able to execute themselves.

Option 5: Pass a Block as Argument instead of Returning a Result.

This is the option that might surprise you: that you can use an additional block *argument* to the parse method instead of a *result*.   Here is the code:

```
method parse(instruction:String) in(code:Block) {
    // split instruction into opCode and operand, and
    // apply code to those two parts.
    def opCode = instruction.subStringFrom 1 size 2
    def operand = instruction.subStringFrom 3
    code.apply(opCode, operand)
}
```

Now the the client code looks like this

```
        parse(inputWord) in { operation, argument ->
            // code that manipulates operation and argument
        }
```

Notice that all of the rest of the program (at least, the parts that use `operation` and `argument`) are now inside the block that forms the second argument to `parse()in().` The two arguments to that block carry the output of parse into the block.   There is a name for this style of programming: *continuation-passing* style.   It's called that because the continuation of the program is passed as an argument to the `pares()in()` method, instead of just appearing after it in the program source.

programming-hints

Updated 1 day ago by Andrew P. Black

followup discussions *for lingering questions and comments*