Worksheet 7: Domain Specific Languages

Updated: 16th October, 2019

1. Fluent API Design

Say we want to provide a flexible way to validate dates, by writing things like this:

```
private Validator v1 =
    date().isAfter("2021-01-01").and().not().isSunday();

private Validator v2 =
    date().isMonday().and()
          .isAfter("2019-12-31").and()
          .isBefore("2021-01-01");
```

The idea is that, once we've created the above `Validator` objects, we should be able to call a method on them to determine whether a given `LocalDate` object[1] is valid. For instance:

```
public void doThing(LocalDate date)
{
    if(v1.isValid(date) || v2.isValid(date))
    {
        ...
    }
}
```

How would you design the `Validator` class to support a fluent API that would make the above code work? You should write enough of the implementing code to demonstrate that your API would work, but you don't need to *actually* make it work.

Hint: it will help to realise that a single `Validator` object represents a list of individual conditions. These individual conditions could themselves be represented using a small collection of other classes with a common superclass.

Note: In fluent APIs, what a method *returns* may have little or nothing to do (directly) with what the method is actually *for*.

2. Parsing a DSL

Imagine that we're developing a network of automated buses, and we want to implement a domain-specific language for specifying the bus times and routes.

[1] `LocalDate` is part of the standard Java API, and simply represents a date (minus a timezone component).

Your task is to implement a parser in JavaCC to read such a language. Here's an example of what it must parse:

```
bus(59) ->
    "Beach Shuttle"
    at 9:30 10:30 11:30 12:00
       13:00 14:00 15:00
    via stops 3 76 44 89 161 32
    free

bus(1234) ->
    "The Hills Loop"
    at 7:15 7:30 7:45 8:05 8:20 8:40 9:00
    via stops 99 97 77 66 145 168

bus(7)  ->  "City Transit"
at
    16:08 16:39 16:55 17:01 17:12 17:28
via
stops
    2 1 5 7 13 119
```

Assume that:

- All whitespace (outside of double quotes) should be ignored.
- There will be any number of `bus` declarations overall, and the various parts of each declaration always occur in the same order.
- The bus name (in double quotes) can contain any number of any characters (including spaces), except a double quote.
- The times are all in 24-hour "hh:mm" format, and there must be at least *one* of them per bus declaration.
- The stop numbers are all pre-defined locations, and there must be at least *two* of them per bus declaration.
- The word "`free`" may or may not be present for each `bus` declaration.

Note: Obtain a copy of `javacc-official-examples.zip` to see some examples of JavaCC grammar files. In particular, the .jj files in the `examples/SimpleExamples/` directory. (These are also available as part of the JavaCC download itself.)

For reference documentation on the JavaCC grammar file format (the technical rules, not the examples), see https://javacc.org/javaccgrm. This documentation occasionally refers to "non-terminals", by which it (basically) means BNF productions.

End of Worksheet