Object Oriented Programming - Exercise 6: Simplified Java Verifier

Contents

1 Goals

1. Implementing concepts learned in class (Regular Expressions).
2. Designing & implementing a complex system.
3. Working with the Exceptions mechanism.

2 Submission Details

- Submission Deadline: **Wedensday, 15/01/2020, 23:55**.
- This exercise will be done **in pairs**.
- Note: In this exercise you may use the following classes in the standard Java distribution: Classes and interfaces from the `java.util`, `java.text`, `java.io` and `java.lang` packages (including sub-package). Specifically, you are required to use `java.util.regexp.Matcher` and `java.util.regexp.Pattern` (for working with regular expressions). Also consider `java.util.function` for existing useful functional interfaces.

  If you choose to use tools not discussed in course material, you must explain in your README file why you chose to use them, and what are the downsides of using them compared to other alternatives (if such exist).

3 Introduction

Regular expressions are used to analyze text by assessing whether a given text string matches a pre-defined pattern. A common setting (which you are now very familiar with) that involves the evaluation of text against a given set of rules is the compilation of programming code. This process involves getting a text file as input, and examining each of its lines to see whether the file is a legal code file, as defined by the language specification.

In this exercise you will implement a Java verifier - a tool able to verify the validity of Java code; it knows how to read Java code and determine its validity, but not to translate it to bytecode.

As Java's syntax is rather complicated, writing a Java verifier is a highly challenging task. To make this task feasible for you, you will in fact implement an s-Java verifier (s stands for *simplified*), which only supports a **very limited** set of Java features (see details below). Your verifier will not have to run any code, it will only output whether or not the input file is a legal s-Java file or not.

4 Input/Output

Your program will receive a single parameter (the s-Java source file) and will run as follows:

```
java oop.ex6.main.Sjavac source_file_name
```

This already demands a certain program structure: a package named `oop.ex6`, containing a package named `main`, which in turn contains a class named `Sjavac` which contains the main

method of the program. You are free to determine which of the other classes you create will belong in the `main` package, and which in other packages under the `oop.ex6` package.

The output of your program is a single digit, outputted using `System.out.println()`:

- 0 – if the code is legal.
- 1 – if the code is illegal.
- 2 – in case of IO errors (see 6.1).

In case of an error, you are also required to print an informative message to the screen using `System.err(error_message)`, describing in general what went wrong. There is no specific format you have to follow here, as it won't be tested automatically. However, very uninformative error messages might result in point reduction. See examples below.

5 *s*-Java specifications

s-Java is a (very) simplified version of Java. In the following, we describe its features.

5.1 General Description

- An *s*-Java file does not interact with other files. That is, you cannot import code from or export to other files. Each file is stand-alone.
- Each *s*-Java file is composed of two kinds of components:
  - *Global variables*, or *members*, are variables shared between methods.
  - *Methods* are general functions. Each method is composed of a list of code lines: defining local variables, giving new values to variables (local or global), calling methods, defining `if/while` blocks and returning (see more details below).
- *s*-Java has no classes. As such, it is appropriate to think of *s*-Java members/global variables as comparable to Java static members, and of *s*-Java methods as comparable to Java static methods.
- Each *s*-Java line is either
  - An empty line, containing only white spaces (spaces, tabs, carriage return, etc. as `\s` is defined to include). It should be ignored.
  - A comment line, in which the first characters are //. No characters, including white spaces, may appear before the //, and any number and kind of characters may appear after the //, and are to be ignored.
  - A code line, which **must** end with one of the following suffixes:
    - ‘;’ - for defining variables, changing variable values, calling methods and returning.
    - ‘{’ - for opening method declarations or `if/while` blocks.
    - ‘}’ - for closing ‘{’ blocks (has to be in its own line).

White spaces may appear before and after any of these suffixes. These suffixes may **not** appear in the next line instead, as in :
`void foo()`
{ // This is not legal!

- Other comment styles like multi-line comments (/* ... */), javadoc comments (/** ... */) and single-line comments appearing in the middle of a line are not supported by *s*-Java . Any appearance of such comments is illegal and should result in a printed value of 1.

- Operators are not supported (`int a = 3 - 5;`, `String b = "OO" + "P";` are illegal).

- Similarly, arrays are not supported in *s*-Java .

5.2 Variables

s-Java comes with a strict set of variable types. It does not support the creation of new types (e.g., classes, interfaces, enums). The language supports the definition of two kinds of variables: global variables and local variables (defined inside a method – see below). Both types of variables are defined in the same way as in java:

type name = value;

- 1 shows the legal *s*-Java `types`.

- *name* is any sequence (length > 0) of letters (uppercase or lowercase), digits and the underscore character (_). *name* may **not** start with a digit. *name* may start with an underscore, but in such a case it must contain at least one more character (i.e., '_' is not a legal name).
  Legal name examples: `'g2', 'b_3', '__', '_a', '____b', '_0', 'a_'`.
  Illegal name examples: `'2g', '_', '2__', '54_a', '3_3__3_b'`.

- *value* can be one of the following:

  – a legal value for *type* (see Table 1); you may assume the following **four** characters will not appear in String or char values: \ ' " ,

  – another existing and initialized variable of the *same type*

5.2.1 Variable declaration

- A variable may be declared with or without an initialization. That is, both `int a;` and `int b2 = 5;` are legal *s*-Java lines.

- A variable declaration must be encapsulated in a single line. For example, the following are **illegal**:

```
int a
= 5;
int g
;
```

Type	*Description*	*Value Format*	*Examples*
int	an integer number	a number (positive, 0 or negative)	`int number1 = 5;` `int num2 = -3;`
double	a real number	an integer or a floating-point number (positive, 0 or negative)	`double b = 5.21;` `double c = 2;`
String	a string of characters	a string of characters	`String s = "hi";` `String a = "i%#";`
boolean	a boolean variable	`true`, `false` or any number (int or double)	`boolean a = true;` `boolean b = 5.2;`
char	a single character	any character (inside single quotation marks)	`char my_cr = 'a';` `char g = '@';`

Table 1: *s*-Java types.

- Multiple variables of the same type may be declared in a single line, separated by a comma. For example, the following lines are **legal**:

```
double a, b;
int i1, i2 = 6;
char c='Z', f;
boolean a, b ,c , d = true, e, f = 5;
String a = "hello" , b = "goodbye";
```

  As mentioned, our tests will not include Strings containing commas, to prevent confusion in cases such as the one shown in the above line.

- There may not be two global variables with the same name (regardless of their types). For example, in the following, given the first line, the second line is **illegal**:

```
int a = 5;
String a = "hello";
```

  However, a local variable can be defined with the same name as a global one (regardless of their types). That is, in the previous example, had the first line been a declaration of a global, the second would have been legal if it was a declaration of a local variable. This is true even if the global variable of the same name was initialized inside the method where the local variable is later declared.

- Methods parameters are considered local variables of the method they belong to.

- Two local variables with the same name (regardless of their type) cannot be defined inside the same block. This also applies to variables with the same name as a method parameter.

- However, two local variable with the same name can be defined inside different blocks, even if one is nested in the other:

```
void foo(int param1){
        int a = 5;
        if (param1){
```

```
            int a = 20;
            boolean param1 = true;
            while (param1) {
                double a = 2.5;
            }
        }
        return;
}
```

In case of multiple variables with the same name, a reference to that name refers to the variable in the most specific scope.

- A variable may have the same name as a method.

5.2.2 Assigning values to variables

- Any variable may be declared using the modifier `final`, which makes it a constant. Such variables **must** be initialized with some value at declaration time: `final int a = 5;`. The modifier should appear before the type of the variable, and not after. In other words, the following is not allowed: `int final a = 5;`.
- A final variable may **not** be assigned a value in subsequent lines (i.e., lines other than it's declaration line).
- Other java modifiers (e.g., `static, public/private`) are **not** allowed (they are not part of the language specification).
- A (non `final`) variable can be assigned with a value after it is created:

```
int a = 521;
...
a = 832;
```

  This applies both to global and local variables. Local variables can only be assigned inside the method they were declared in (including any scopes nested in it). Global variables may be assigned multiple times both inside and outside a method.

- A local variable cannot be used in any way (in an assignment or a call to a method) before it is assigned a value (you don't have to check that the variable initialization is reachable, e.g., inside an `if (false) {}` block). In other words, if a local variable is not initialized in its declaration, the code is only legal if:
  - The next line **in which it appears** is an assignment of a value to that variable.
  - It is never used.
- A variable `a` (global or local) can be assigned with another (global or local) variable `b` of the same type. Additionally, a `double` can also be assigned with an `int`, and a `boolean` can also be assigned with an `int` and a `double`. For example:

```
int a = 5;
double b = a;
```

5.2.3 Referring to variables

- Unlike a local variable, a global variable can be referred to by methods appearing before the line in which it was assigned a value or declared. Other global variables may only be assigned a value using a global variable if it was already declared and assigned.
- In a case of an un-initialized global variable (meaning it is not assigned a value anywhere outside a method), all methods may refer to it (regardless of their location in relation to its declaration), but every method using it (in an assignment, as an argument to a method call) must first assign a value to the global variable itself (even if it was assigned a value in some other method).
- Accessing variables declared in a more inner scope is illegal.

5.3 Methods

s-Java methods are a simple version of Java's methods definition:

$$\texttt{void}\ \textit{method\_name}\ (\ \textit{type}_1\ \textit{parameter}_1,\ \textit{type}_2\ \textit{parameter}_2,\ \ldots,\ \textit{type}_n\ \textit{parameter}_n)\ \{$$

where:

- *method_name* is defined similarly to variable names (i.e., a sequence of length > 0, containing letters (uppercase or lowercase), digits and underscore), with the exception that method names must start with a letter (i.e., they may not start with a digit or an underscore).
- *parameters* is a comma-separated list of parameters. Each *parameter* is a pair of a valid type and a valid variable name, **without** a value.
- Only `void` methods are supported.

After the method declaration, comes the method's code. It may contain the following lines:

- Local variable declaration lines (as defined in 5.2)
- Variable assignment lines (as defined in 5.2).
- A call to another existing method. Any method `foo()` may be called, **regardless** of its location in the file (i.e., before or after the definition of the current method). The syntax of calling is the java syntax:

  $$\textit{method\_name}\ (\textit{param}_1, \textit{param}_2, \ldots, \textit{param}_n);$$

  where *method_name* is an existing method and $\textit{param}_1, \textit{param}_2, \ldots, \textit{param}_n$ are variables or constants (`3, "hello"`) of types agreeing with the method definition (though a `double` parameter can accept an `int` argument and a `boolean` can `int` and `double`). Calling a method with an incompatible number of arguments, wrong types or uninitialized variables is **illegal**.
- An `if/while` block (see below).

- A `return` statement. Return statements may appear anywhere inside a method, but must also appear as the last line in the method's code (you do not need to check for unreachable code due to previous `return` statements); this is different from Java where a method can also simply end without a `return` statement. The `return` statement should not return any value, in accordance with the method's declaration. The format is:

```
return;
```

Note the following:

- Method parameters may be final. For example:

  ```
  void foo(final int a) {
  ```

  In this case, it is still legal to call the method with a non-final variable. It only means that the value of the parameter may not be changed inside the method.

- You may change the value of a non-final parameter inside the method's code.

- Methods must end with a line containing the single token '}' (which comes right after the `return` line as presented above).

- Recursive calls are allowed. I.e., a method may call itself. You are not required to check for infinite loops/recursions in the code.

- Method overloading is not supported: no two methods with the same name may exist.

- The order in which methods appear in the code is meaningless. Methods may also appear before/after/between global variable declarations.

- A method may **not** be declared inside another method.

- Method calls may only appear inside a method, and **not** in the global scope.

5.4 `if` and `while` Blocks

`if` and `while` blocks may only appear **in methods**, and are defined in the following way:

```
if (condition) {
        ...
}
while (condition) {
        ...
}
```

where `condition` is a boolean value (as defined in table 1), so it is either:

- One of the reserved words `true` or `false`.

- An initialized `boolean`, `double` or `int` variable.

- A `double` or `int` constant/value (e.g. `5, -3, -21.5`).

Also, multiple conditions separated by AND/OR may appear (e.g., `if ( a || b || c) {` is a legal statement).

Notice the following:

- The `||` and && operators must be placed between two boolean expressions, so for example `if(|| a)` is illegal, and so is `if(a || && b)`.
- `if/while` blocks may contain the same type of lines as methods (that is, variable declaration, method calls, etc. See 5.3).
- You are not required to support conditions containing brackets, like
  `if (( a || b) && c ...) {`
  ...
  }
- Much like methods, `if/while` blocks must start with a '{' token (which should be at the end of the condition line) and end with a '}' token (which should be in a line of its own).
- `if/while` blocks can be nested to a practically unlimited depth (i.e. you should support a depth of at least `java.lang.Integer.MAX_VALUE`): `if` inside `while` and vise versa.
- You are not required to support `else if`, `else` blocks, `do/while` loops, `for` loops or `switch` statements. That is, the words `else`, `do`, `for` and `switch` are not part of the *s*-Java specification.

5.5 General Comments

- A `main` method is not required in *s*-Java . Of course a method called `main` may be defined, much like any other method.
- Naming variables or methods with reserved words (e.g `int`, `if`...) is not covered by the *s*-Java specification, and so you can handle such cases as you wish. (For example: `int double = 6;`)
- White spaces are allowed anywhere in the code, and are to be ignored. This **excludes** white spaces inside variable names, values or reserved words (e.g., the following line is **illegal**: `i n t  a = 5;`), and in the start of comment lines, so the following line is illegal:
  `" \\comment after spaces"`
- On the other hand, white spaces are required to separate between:
  - A method's return type (only `void` in *s*-Java) and name (e.g., `voidfoo(){` is illegal).
  - A variable's type and name (e.g., `inta;` is **illegal**).
  - A `final` modifier and a variable type (e.g., `finalint a;` is **illegal**).

  Zero or more white spaces may appear between any other input tokens (this includes before/after parentheses, '=', ';','{' and '}' characters, etc.).
- Each line of code must appear in a single line, and not broken into several lines. Also, no two statements may appear in the same line.

- Packages, as well as exceptions, are not supported in *s*-Java . That is, the words `package`, `try`, `catch` and `finally` are not part of the *s*-Java specification.
- The reserved keywords in *s*-Java are:
  - Types: `int, double, boolean, char, String`.
  - Other: `void, final, if, while, true, false, return`.

6 Important Requirements

6.1 Error Handling

As in *ex5*, you are required to use the exceptions mechanism to handle general errors of your program (e.g., an IOException caused by an illegal file name). As noted earlier, in such cases you should print the value 2 and an understandable error message.

Regarding the main task (deciding whether or not the file is a legal code file), error handling is a bit tricky. Supposedly, all the information many of your methods have to provide in this exercise is a single bit – whether or not some part of the code is legal. In this case, it is tempting to define boolean methods that return true or false according to whether the code is legal or not. However, some of the benefits of the exceptions mechanism might come in handy here. We advise you to consider using this mechanism in this exercise. **Regardless, report in your README file how you handled *s*-Java code errors in this exercise, and why you chose to do so.**

6.2 Object Oriented Design

As in previous exercises in this course, your program should follow the object oriented design principles. The working process should be similar to the following:

- Try to divide your program into small, independent units.
- Try to think of several design alternatives for each unit, consider the pros and cons of each alternative, and select the one you think is best.

You are required to specify your design, as well as your thinking process and the alternatives you ruled out, in your README file.
In addition, you should address the following points in your README file:

- How would you modify your code to add new types of variables (e.g., `float`)?
- Below are four features your program currently does not support. Please select **two of them**, and describe which modifications/extensions you would have to make in your code in order to support them. Please briefly describe which classes you would add to your code, which methods you would add to existing classes, and which classes you would modify. You are **not required** to implement these features.
  - Classes.
  - Different methods' types (i.e `int foo()`) .
  - Using methods of standart java (i.e `System.out.println`).
  - Inheritage of files (just like class inheritage, so `virtual` functions can appear, and a sub-file may use a method from a parent file).

6.3 Regular Expressions

One of the main goals of this exercise is to practice the use of the regular expression mechanism in Java (and in general). You are therefore required to make extensive use of it in your program. However, you are allowed (and encouraged!) to use other text analyzing mechanisms in your code (e.g., `String` class methods such as `substring(), charAt()`, etc.). The general rule of thumb is that you should use regular expressions whenever it makes your life easier, and not when it makes it harder.

You are encouraged to use online tools to test your regular expressions (i.e regex101), Please notice the special escaping of java, for example a digit in regex101 will be denoted by '\d' while in Java it will be '\\d'

In your README file, please describe two of the main regular expressions you used in your code.

7 Submission Requirements

7.1 README

The README instructions from previous exercises apply here as well. Please describe your design and focus on non-trivial decisions you made. **The README file should also include answers to the questions from Sections 6.1, 6.2 and 6.3.** This exercise should be submitted in pairs, unless you notified the staff otherwise. To submit as a pair, **the first two lines of the README must be your CS usernames**.

To submit the exercise as a pair, **only the student whose CS username is the first line of the README should submit the exercise**.

When you submit the exercise, make sure that both your usernames appear in the output of the presubmission tests.

7.2 Automatic Testing

A file called `ex6_files.zip` can be found in the course website. The zip contains a list of 90% of the *s*-Java files on which your program will be tested. Additionally, the automatic tests will produce error messages containing the file number of failed tests. For example, you may receive the following error:

```
### return value error, when ran with arguments test011.sjava :
boolean global test no value
```

This means that your program printed a different value (0,1 or 2) than the school solution when tested with the file `test011.sjava`. The second line gives a short description of that specific test; in this case, the test checks how your verifier handles a declaration of a boolean global variable with no assigned value.

You can run the presubmission script directly from the shell on your `.jar` file by typing

```
~oop/bin/ex6 myJarFile.jar
```

7.3 Submission Guidelines

As in previous exersices, you should submit a file named `ex6.jar` containing all `.java` files of your program, as well as your README file. Files should be submitted in the directories of

their original packages. No `.class` files should be submitted. Remember that your program must compile without any errors or warnings.

7.4 School Solution

The school solution may be found under `~oop/bin/ex6school`. You are strongly advised to experiment with it before starting to work on your design, in order to get a feeling of how your program should behave on different inputs. You can run it on any `.sjava` file using the following command:
`~oop/bin/ex6school file.sjava`

Good Luck!