SOFTWARE TESTING COURSEWORK SPECIFICATION

February 2022

To be read in conjunction with the coursework description.

1 Specifications for tasks 1 and 2

1.1 Overview

The command line option parser is developed to process command line options stored in an input string and output a set of value pairs containing information retrieved from the input string.
The user can first define a set of options:

- Option name `filename`, with a String value type.
- Option name `output`, with a shortcut `o` and a String value type.

After the options are defined, the parser is able to process the following example input of command line options:

```
--filename 1.txt -o 2.txt
```

In this example, the prefix `--` indicates using the name of the option while `-` indicates using the shortcut of the option.
The parser will get the following value pairs:

```
filename = "1.txt"
output = "2.txt"
```

1.2 Parser Initialisation

The parser is initialised using

```
Parser parser = new Parser();
```

There is no further specification regarding class initialisation.

1.3 Add options with a shortcut

```
void addOption(Option option)
void addOption(Option option, String shortcut)
```

This method is for adding a new option that has a shortcut.

Example:

```
parser.addOption(new Option("output" , Parser.STRING), "o"));
parser.addOption(new Option("optimise" , Parser.BOOLEAN, "O"));
```

Parameter list:

`String` *option* : An option object.
`String` *shortcut* : a shortcut of this option.
`Option` is a custom type, containing three fields: `name` of type String, `value` of type String and type of type `Type`, which is a enumeration containing the following choices:

- `Type.INTEGER`
- `Type.BOOLEAN`
- `Type.STRING`
- `Type.CHAR`
- `Type.NOTYPE` (Not a valid type, used for corner cases handling).

Notice that Option does **not** contain the shortcut value. The reason is it is handled separately, to allow multiple shortcuts to be assigned to the same option. More on this, on the specifications section below.

Return value:

This method does not have a return value.

Specifications:

1. Adding an option with the same name as an existing option will lead into overwriting the old type only (as name is the same) in the already existing option object held in the options Map held internally in the `OptionMap` class. The old Option object is preserved in this case, which means that previously defined shortcuts will be able to access it and its old value is also preserved.
2. Name and shortcut of options should only contain digits, letters and underscores. Digits cannot appear as the first character. A runtime (`IllegalArgumentException`) exception is thrown otherwise.
3. Option names and shortcuts are case-sensitive.
4. An option can have a shortcut that is the same as the name of another option. For example, the user can define an option whose name is `output` with a shortcut `o` and another option whose name is `o`. When assigning values to these options, `--output` and `-o` is used for the first option and `--o` is used for the second option.
5. Boolean options are assigned to `true` using any value except 0, false (case insensitive) and without having a value - all three cases will lead to `false`. For example, the option `optimise` is set to false by `-O`, `--optimise`, `-O=false`, `-O=0`, while it is set to true by `-O=true`, `-O=1`, `--optimise=Anything` or `-O 100`.

1.4 Add options without a shortcut

```
void addOption(Option option);
```

This method is for adding a new option that does not have a shortcut. You will still be able to add a shortcut to the option, using the `setShortcut(...)` method, described in Section 1.8).

Example:

```
parser.addOption(new Option("output" , Parser.STRING));
parser.addOption(new Option("optimise" , Parser.BOOLEAN));
```

Parameter list:

`String` *option* : The Option object to add.
`int` *value_type* : type of the expected value. Choices of the parameter are grouped in the Enumeration type Type and include:

- `Type.INTEGER`
- `Type.BOOLEAN`
- `Type.STRING`
- `Type.CHAR`
- `Type.NOTYPE` (Not a valid type, used for corner cases handling).

Return value:

This method does not have a return value.

Specifications:

Same as specifications of the first add function.

1.5 Parse command line options:

```
int parse(String commandLineOptions);
```

Parse the string containing command line options.

Example:

```
parser.parse( "--input 1.txt  --output=2.txt" );
parser.parse( "-O" );
```

Parameter list:

`String` *commandLineOptions* : command line option string. Can be blank but not empty or null.

Return value:

This method returns 0 if the parsing succeeds and other values if the input is not valid or the parser fails.

Specifications:

1. The prefix `--` is used for the full name of the option.
2. The prefix `-` is used for the shortcut version of the option.
3. Assigning a value to the option can either be using a `=` or a space. That is, `option=value` and `option value` are both valid.
4. The user can use quotation marks, single or double, optionally around the value. `option="value"` , `option='value'` and `option=value` are all valid and result in the same effect.
5. As quotation marks can be used to decorate values, they can still be part of the value if the user wants. If single quotation marks are used, double quotation marks in the value string are viewed as part of the string and vice versa. For example, `option=' value="abc" '` is valid and the value of the option is `value="abc"`.
6. If the user assigns values to the option (even using its shortcut) multiple times, the value taken by the option is from the last assignment.
7. The user does not need to provide values for every option. For the options that are not assigned a value using this function, a default value (described in Section 1.6) is stored.
8. This method can be used multiple times as shown in the example above. After these two function calls, three options are assigned to values and can be retrieved by the following methods.
9. When requesting the integer from a character, you obtain its value in ASCII.
10. Values can contain any character, but are restricted NOT to contain the following reserved keywords (including brackets): `{D_QUOTE}`, `{S_QUOTE}`, `{SPACE}`, `{DASH}` and `{EQUALS}`.
11. Note that we utilize an enum for types - as only the types`String`, `Integer`, `Character` and `Boolean` are valid. `NOTYPE` is not considered a valid type, and exists for verification purposes.

    These are important keywords for the parsing procedure, and using them as strings will lead to undefined behavior.

1.6 Retrieve information

```
int getInteger(String optionName);
boolean getBoolean(String optionName);
String getString(String optionName);
char getCharacter(String optionName);
```

These methods retrieve the value of an option.

Example:

```
String output_file_name = parser.getString("output");
boolean optimise = parser.getBoolean( "O" );
```

Parameter list:

`String` *option* : the full name or shortcut of an option.

Return value:

These methods return the value of the corresponding option.

Specifications:

1. If not any dash is added preserving the option name, the order of search is full name of options first and then shortcut. For example, if `o` exists as a full name for an option and a shortcut for another option, this function returns the value of the first option.
2. If the option is defined but is not provided a value, a default value is used: 0 for `INTEGER`, `false` for `BOOLEAN`, an empty String `""` for `STRING` and `'\0'` for CHARACTER.
3. If the option is not defined, a runtime exception is thrown. You can use `optionOrShortcutExists(...)`, `optionExists(...)` and `shortcutExists(...)` to check if an option/shortcut exists.
4. if a double dash (`--`) is provided before the variable, then the method will look only for options. Call example: `getBoolean("--optimise")`.
5. if single-dash (`-`) is provided before the variable, then the method will look only for shortcuts. Call Example: `getBoolean("-O")`.
6. Note: it is important to consider that, different types of options with the same values initialized will give different values occasionally. for instance an `INTEGER` initialized with the "false" value will return 0 when `getInteger(...)` is invoked for that option, but calling the same method for that value on a `CHARACTER` will return 102, the ASCII number of the first letter of `false`. You have to be careful for such considerations.

1.7 Check information

```
boolean optionExists(String optionName);
boolean shortcutExists(string optionName);
boolean optionOrShortcutExists(String optionName);
```

These methods check if an option, shortcut, or any of those two exists.

Example:

```
parser.addOption(new Option("output" , Parser.STRING));
System.out.println(parser.optionExists("output")); // Prints true
```

Parameter list:

`String` *option* : the full name / shortcut of an option.

Return value:

A boolean value, indicating if a variable exists. Their usage is straightforward.

1.8 Modify information

```
void replace(String variables, String pattern, String value)
void setShortcut(String option, String shortcut)
```

Example:

```
// Variables are previously defined, with values set.
parser.addOption(new Option("opt1", Parser.STRING));
parser.addOption(new Option("opt2", Parser.STRING));
parser.parse(--opt1=OldText --opt2=OldText2)
parser.replace( "opt1 opt2", "Old", "New");
parser.setShortcut("opt1", "o");

// Prints "NewText"
System.out.println(parser.getValue("o"));
```

Specifications:

1. `replace(...)` is a method that applies bulk replacement to the values of the options, wherever possible. The function takes 3 arguments: `variables`, which are previously defined option names or shortcuts, space-separated, `pattern`, which is the pattern to search for (also supporting regex expressions) and `value` is the value to replace.

   Note that the same rule with getter methods apply: if double dash is specified before a variable, only the options will be considered for replacement. If a single-dash is specified, only shortcuts will be considered. However, if no dash is specified, the `replace(...)` will attempt to replace values both in options **and** shortcuts.

2. `setShortcut(...)` is a method that adds a new shortcut to an option, following its initial declaration. It takes 2 arguments. `option`, which is the option name, and `shortcut`, which is the new shortcut name. Note that, using this method, we are able to assign multiple shortcuts related to an option. Adding an existing shortcut, replaces an older one, "dereferencing" the old option, in a sense.

2 Specifications for task 3

The system has to also support the bulk insertion of options. This can be done via the following ways:

```
void addAll(String options, String shortcuts, String types);
```

Parameter list:

`options:` The first argument represents a String containing a list of option names or an option group, space-separated.
`options:` The second argument represents a String containing a list of option shortcuts or a shortcut group that correspond to the options given in the first argument, space-separated.
`options:` The third argument represents a String containing a list of types that correspond to the options given in the first argument, space separated.

Return Value:

The method has no return value.

Example:

```
Parser parser = new Parser();

// Bulk initialization.
parser.addAll("option1 option2 option3 option4","o1 o2 o3 o4",
              "String, Integer, Boolean, Char");

// Initialize without shortcuts. Ignore "String" type.
parser.addAll("option5", "Integer String");

parser.parser("-o1=test -o2=33 -o3=true --option4=a --option5=12");

// Prints 'a'.
System.out.println(parser.getChar("option4"));
```

In this scenario, this initializes the options in sequence:

- `option1` has shortcut `o1`, type `Parser.String` and value `"test"`.
- `option2` has shortcut `o2`, type `Parser.Integer` and value `33`.
- `option3` has shortcut `o3`, type `Parser.Boolean` and value `true`.
- `option4` has shortcut `o4`, type `Parser.Char` and value `'a'`.
- `option5` has no shortcut, type `Parser.Integer` and value `12`

Specifications:

1. Options are initialized and matched with shortcuts and types following the same order along the lists in the arguments.
2. Lists contain space-separated options/shortcuts/types. The parser rules for option/shortcut naming apply here as well.
3. Extra spaces are omitted.
4. In similarity with `"parser.addOption(...)"` method, the `"shortcuts"` argument is optional.
5. If more options than shortcuts are given, then the options that exceed the number of shortcuts will have no shortcut.
6. If less options than shortcuts are given, then the extra shortcuts are omitted. Same applies if more types than options are given.
7. If less types than options are given, then the remaining options will have the last type in the sequence. For example:

   ```
   parser.addAll("option1 option2", "o1 o2 o3" "String");
   ```

   generates `"option1"` and `"option2"` variables with shortcuts `"o1"` and `"o2"`, all of type String.

Group Initialization

The method will support option and shortcut group initialization. By groups, we mean syntactic sugar for generating a number of options or shortcuts by specifying a range at the end of its declaration. One is able to initialize groups (or ranges) of variables using the `-` character. The last range element of the group is inclusive.
For example:

- `option1-3` will generate `option1, option2 and option3` variables.
- `optiona-c` will generate `optiona, optionb and optionc` variables.
- `optionA-C` will generate `optionA, optionB and optionC` variables.

In code:

```
// Option group initialization.
// Notice that shortcut range covers both groups.
parser.addAll("option7-9 optiona-c optionA-B", "o7-12" "Integer String");

// option7-9 group consists of Integers
parser.parser("--option7=1 --option8=test --option9=test2");
// optiona-c group consists of Strings
parser.parser("--optiona=Test --optionb=TestOption --optionc=TestAgain");
// optionA-B group consists of Strings, as the last type defined.
// They have no shortcuts.
parser.parser("--optionA=Test --optionB=TestOption");

// Prints 2.
System.out.println(parser.getString("option8"));
```

For these initialization actions, all the above specification rules apply, plus the following:

1. Groups apply in options and shortcuts, if any shortcuts are provided.
2. Group must have the form:

   ```
   base => underscore/letters/characters/numbers
   range_value1/2 => letters/characters/numbers
   [base][<range_value1>-<range_value2>]
   where range values of the same group.
   ```

   or:

   ```
   (([A-Za-z0-9_])+(([A-Z]-[A-Z]+)|[a-z]-[a-z]+|[0-9]-[0-9]+))
   ```

   in regex.
3. Both range values are inclusive.
4. Range values are strictly within the same type: number to number, lowercase to lowercase letter, or uppercase to uppercase letter. Combination of groups, such as `aA-z` is invalid and should be omitted.
5. If a group has an invalid form, then it is ignored altogether from the sequence, along with its respective shortcut and type, if any.

6. Groups are defined with letters (uppercase/lowercase) and numbers only.

7. When a type is defined on the third argument, it is related to the entire corresponding group. In the example above, `option7-9` was associated with the `Integer` type, while `optiona-c` with the `String` type.

8. In order to generate a group the range will be considered by taking the last character before hyphen as a start, and all the characters after hyphen as the end.

   For example:

   ```
   parser.addAll("g129-11");
   ```

   generates the options `g129, g1210, g1211` and **not g129, g130, g131**, as the range was `9-11`.

9. the range will always be specified at the end of the group. If this is not the case `(e.g., g1234-7ab)` the group is invalid and should be ignored altogether, along with its shortcuts and types.

10. Decreasing ranges are allowed. For instance:

    ```
    parser.addAll("g125-2");
    ```

    generates the options `g125, g124, g123 g122`.

11. Multiple option groups can correspond to one shortcut group, and vise versa. For example:

    ```
    parser.addAll("option1-2 option3-4", "o1-4" "Integer");
    ```

    declares `option1, option2, option3, option4` variables with `o1, o2, o3, o4` shortcuts and type `Integer`.

12. The only allowed values for types within the last string, are `String`, `Integer`, `Character` and `Boolean`, matching the provided enum. Any other values should throw an Exception.