Whitespace

Whitespace is a programming language, where code is written using exactly three characters, all of which are whitespace:

- The space character: as a Java String, " "
- The tab character: as a Java String, "\t"
- The line-feed character: as a Java string, "\n"

All other characters are ignored, and assumed to be comments.

In this task, you will produce an **interpreter** for Whitespace – a Java program that can run Whitespace programs.

A note on Stacks

Ignoring the syntax for a moment, Whitespace is a **stack-based language**. A stack is a data structure where, in terms of arrays:

- An item can be **pushed** onto the stack: added to the end of the array
- An item can be **popped** off the stack: removed from the end of the array
- We can **peek** at the item on the top of the stack: look at the item at the end of the array.

One way to implement a stack in Java is to use a LinkedList: to push an item onto the stack use, addLast(); to pop an item from the stack, use removeLast(); to peek, getLast().

The instructions in stack-based programming languages work by manipulating the stack. For instance, to add together the numbers 3, 4 and 5:

- Push 5 onto the stack. (The stack now looks like {5}).
- Push 4 onto the stack. (The stack now looks like {5,4}).
- Push 3 onto the stack. (The stack now looks like {5,4,3}).
- Use the 'add' command. This pops two items off the stack (4 and 3); adds them together; and pushes the answer onto the stack. The stack now looks like {5,7}.
- Use the 'add' command again, which this time adds 5 to 7. The stack now looks like {12}.
- If we peek at the stack, we now get the answer: 12.

i) Whitespace interpreter step 1: arithmetic operations

A reasonable first step for writing a Whitespace interpreter is to write the stack arithmetic operations: methods that take a stack; pop off two values; add/subtract/etc. them; and push the answer onto the stack.

For instance:

```
public static void addStack(LinkedList<Integer> stack) {
	// TODO: pop two numbers off stack, keeping them in temporary variables
	// add them together, push the answer onto stack
}
```

There are five arithmetic operations in Whitespace: addition, subtraction, multiplication, division and modulo (%). Note that the first item pushed onto the stack is on the left-hand side of the operator: push 3, push 4, then subtract, should do 3 - 4, not 4 - 3.

You should then be able to test these with code such as:

```
LinkedList<Integer> s = new LinkedList<>();
s.addLast(10);
s.addLast(4);
s.addLast(5);
addStack(s);
subtractStack(s);
System.out.println(s.getLast());
```

...which should print out 1 (add 4 and 5; take that away from 10).

ii) Reading whitespace code from disk

Write some code that will read a file from disk, and store the file in a single long String. Remove from the String all characters that *are not whitespace* – as these are comments, not code.

Whatever you do, **do not use a Scanner**. Scanners skip over whitespace characters, which is exactly what you don't want to do here: you'd be left with all comments, and no code. You don't want anything that splits the file by newlines: you want the raw data. If in doubt, look up **`DataInputStream`**.

Test your code by reading in add2and3.ws from KEATS. It should be 16 characters long, and equal to the Java string " \t \n \t\t\n\t "

iii) Basic operations in Whitespace

An interpreter for Whitespace essentially needs to work through the String containing the code, character by character – starting at character 0 – do whatever the next instruction is in the code, by looking at what the characters are. Full details of the language specification are at http://compsoc.dur.ac.uk/whitespace/tutorial.php .

The first sort of instruction we'll look at are **stack manipulation** instruction: these are denoted by the current character being a space. The simplest of these is to push a number onto the stack, denoted by the *next* character being a space, followed by the number itself. The number is encoded as follows:

- The first character is either a space for positive numbers, or tab for negative numbers. *(For those who took 4CCS1CS1 in the first term: Whitespace doesn't use two's complement.)*
- After that, the number is written in binary, where space = 1 and tab = 0
- The number finishes at a newline character, '\n'.

For instance, to push 2 onto the stack: it's positive, and in binary written as 10, so the code will read:

- " ": it's a stack manipulation instruction
- " ": ...to push a number onto the stack
- " ": it's a positive number
- "\t ": tab-space is 10 in binary
- "\n": newline, to mark the end of the number

Or, in full, " \t \n". You'll notice this is the start of the code in 'add2and3.ws', quoted on the previous page. The next characters in that string read " \t\t\n" – you should be able to figure out what that does.

Implementing 'push onto stack'

Make a class State that contains the state of Whitespace program execution:

- A string `program`, to contain the code. The constructor for State should take the program to its constructor, and use it to define this variable.
- A stack (initially empty), in a variable called `stack`.
- The current position of program execution within that string (initially 0), in a variable `currentPosition`.

In your main method, read in a program from disk, and create a State object for that program. Then, write a while loop that finishes if `currentPosition` is the length of `program`. Within the while loop, you will write code that looks at the character at position `currentPosition`

within `program`, and acts appropriately.

To start with, if this character is a stack manipulation instruction (a space) to push a number onto the stack (i.e. followed by another space), call a suitable helper method to push the number onto the stack.

Now that numbers are on the stack, you can use the methods you wrote in part (i) to perform arithmetic operations on them. Arithmetic operations in whitespace are represented by the current character being tab, and the next character being space: the Java String "\t ". The exact arithmetic operation to perform is represented by the next two characters after that: space space is addition; space tab is subtract; etc. See the language specification for details.

Implementing arithmetic

Extend your main loop so that if the current character is tab, followed by space, it calls a helper method that looks at the two characters after that to see what sort of arithmetic operation to perform, and call the appropriate arithmetic method you wrote in part (i)

To test your code works: add a line after your main loop to print the value at the top of the stack, and run 'add2and3.ws'. It should print out the value 5.

iv) Basic IO in Whitespace

Whitespace has two basic IO operations:

1. Pop the number at the top of the stack, print it out as an ASCII character
2. Pop the number at the top of the stack, print it out as a number

Both IO operations begin with the characters tab then newline: "\t\n". If the following characters are space-space, print as a character; if they are space-tab, print as a number.

Implementing basic IO

Extend your main loop to handle IO. If the current character is tab, followed by newline, then the program is doing IO: look at the next two characters after that to find out whether to print as a number or character, then pop and print the value at the top of the stack accordingly.

To test this, remove the line after your main loop to print the value at the top of the stack, and use add2and3thenprint.ws from KEATS: this is the same as the earlier example, but followed by "\t\n \t": print out the number at the top of a stack as a number

Note that IO is destructive: it actually pops the value from the stack. Sometimes we want the value on the stack to still be there after printing it. For this, Whitespace provides another stack manipulation instruction: duplicate the top item on the stack.

Implementing duplicate

Extend your main loop so that if the next character is a stack operation (space) to duplicate the top item on the stack (newline-space), the item on the top of the stack is duplicated.

v) Labels

How do we do loops in Whitespace? A combination of things: a **label** and a **jump**, known in Whitespace as **flow-control** instructions. Labels are used to mark positions in the code (e.g. the start of where we want to loop), and jumps are used to move the current execution point of the code (the current character the main loop is looking at) to one of these labelled positions.

In Whitespace, if the current character in the String is a newline "\n", then what follows is a flow-control instruction. If the next two characters are space-space, this means the flow-control instruction is a label. The label itself is then an arbitrary sequence of spaces and tabs, followed by a newline. This sequence of spaces and tabs is the **name** of the label. For instance "\n \t \n" is a flow construction, to define the label with the name " \t " (space-tab-space).

Handing labels

Extend your main loop so that if a label is encountered, its position in the code is remembered. Use a Map for this, as covered in lecture 6 of PRP: the name of each label (a String, comprising space and tab characters) maps to an Integer – the current position in the code.

The simplest jump in Whitespace is an **unconditional jump**: represented the flow-control instruction space-newline. It is followed by the name of a label (and a newline), and changes the current position in the code to wherever that label is.

Handling unconditional jumps

Extend your main loop so that if an unconditional jump is encountered, the current position of the code is changed to the appropriate entry from the Map. **Include error handling** – don't allow jumps to labels that aren't found in the map.

To test your code, run evennumbers.ws – this is an infinite loop, printing out all the even numbers. Turning whitespace into something that can be seen, it is written as:

0: [space][space] [space][tab][space][newline] – push +2 onto the stack

6: [newline][space][space] [space][space][space][newline] – define the label space-space-space

13: [space][newline][space] – duplicate the value at the top of the stack

16: [tab][newline] [space][tab] – print out the value at the top of the stack, as a number

20: [space][space] [space][tab][space][tab][space][newline] – push +10 onto the stack

29: [tab][newline] [space][space] – print this character (character 10 is a newline)

33: [space][space] [space][tab][space][newline] – push +2 onto the stack

39: [tab][space] [space][space] – arithmetic: add together the top two numbers on the stack

43: [newline][space][newline] [space][space][space][newline] - jump to the label space-space-space

To actually do conditional statements (if...) or conditional loops (for, while, …), Whitespace does **conditional jumps**. These pop an item from the stack, then only jump to the given label if either:

- The item was zero: the flow-control instruction tab-space; or,
- The item was negative: the flow-control instruction tab-tab.

As with unconditional jumps, these are followed by a label of tab- and space-characters, and then a newline.

Handling conditional jumps

Extend your main loop so that if an conditional jump is encountered, the value at the top of the stack is popped, and if it is zero/negative (as appropriate) the current position of the code is changed to the appropriate entry from the Map. Otherwise, nothing happens.

To test your code, run evennumbersto20.ws which is equivalent to:

[space][space] [space][tab][space][newline] – push +2 onto the stack

[newline][space][space] [space][space][space][newline] – define the label space-space-space

[space][newline][space] – duplicate the value at the top of the stack

[tab][newline] [space][tab] – print out the value at the top of the stack, as a number

[space][space] [space][tab][space][tab][space][newline] – push +10 onto the stack

[tab][newline] [space][space] – print this character (character 10 is a newline)

[space][space] [space][tab][space][newline] – push +2 onto the stack

[tab][space] [space][space] – arithmetic: add together the top two numbers on the stack

[space][newline][space] – duplicate the value at the top of the stack

[space][space][space][tab][space][tab][tab][space][newline] – push 22 onto the stack)

[tab][space][space][tab] – arithmetic: subtraction

[newline][tab][tab][space][space][space][newline] – if negative, jump to space-space-space

vi) Doing labels properly

Unfortunately, with labels, there's a snag that the approach described so far cannot handle. It is permissible to jump to a label that has not yet been reached. Your code so far will catch this an error, but in fact it's a feature. To fix this, you need to add a preprocessing mode, so that the code is processed – to find out where the labels are – but nothing is actually executed.

I would suggest adding a boolean variable `preprocessing` to the State class. Then, in all your code, if `preprocessing` is true, don't call any methods on the Stack, do IO, or jump anywhere:

- Every time there is an instruction to push a number onto the stack, find what the number is, but don't actually put it onto the stack – as that would mean calling a method on the Stack.
- For arithmetic, work out what sort of arithmetic operation it is, but don't actually do the arithmetic – as then you'd have to call methods on the Stack.
- For IO operations, find out whether to print a character or a number, but don't actually pop the value off the stack and print it. (blah blah methods on the Stack blah blah)
- For jump instructions, find out what the label is, but don't actually do the jump, and don't actually report an error if the label cannot be found.
- For labels, **put the labels into the map**. This is all you actually do.

Then, put a 'for' loop around your main loop, so that first, it runs through the program with `preprocessing` set to true; then it runs through it with `preprocessing` set to false.

```
State exec = new State(code); // make the State, as before
for (int pass = 0; pass < 2; ++pass) {
     // reset current position to 0 to go from the start
     exec.currentPosition = 0;

     if (pass == 0) {
          exec.preprocessing = true;
     } else {
          exec.preprocessing = false;
     }
     //  your main loop as before goes here
}
```

vii) Missing stack manipulation instructions

Implement these:

Swap: space (to denote stack manipulation) followed by newline-tab, should swap the two items on the top of the stack with each other.

Discard: space followed by newline-newline should pop the item off the top of the stack

Copy: space followed by tab-space followed by a number *n* followed by a newline, should copy item *n* from the stack, and push it onto the stack. Element 0 is the top of the stack, element 1 is the one after that, and so on. For instance, space-tab-space-space-tab-tab-space-newline copies element tab-tab-space = 110 = 6 from the stack, to the top.

Also implement one extra flow-control instruction:

End: newline (to denote flow-control) followed by newline-newline, should exit the program.

In all cases, if `preprocessing` is true, don't actually do anything: swap doesn't do a swap; discard doesn't do a pop; copy finds out what *n* is but then doesn't do a copy; end just carries on as if it wasn't there.

To test your code, download count.ws given as an example at the bottom of the Whitespace tutorial, and try to run it.

viii) Other missing instructions

There are some other missing parts of Whitespace. I suggest you implement these in the following order:

a) The 'slide' stack manipulation

Pop the item off the front of the stack into a temporary variable. Pop *n* items off the stack and ignore them. Push the temporary variable back on. (Or, if `preprocessing` is true, find out what *n* is, then do nothing.)

b) Heap access

A heap is a collection of variables, each with a unique numeric ID (its address). You can implement this as a map from Integers, to Integers: the keys of the map are addresses; the corresponding value is the value of the variable. (If `preprocessing` is true, these do nothing.)

c) Read IO for heap variables

These do IO from the keyboard. There are two versions:

- One reads a single character (without waiting for a newline)
- One reads a number (waiting for a newline to signify the end of the number)

In both cases, the data read from keyboard is stored on the heap – to get the address to use, pop the value off the top of the stack.

(If `preprocessing` is true, these do nothing.)

d) Call/end a subroutine

Subroutines in Java are known as methods. In Whitespace, they're defined by giving them a label, then using 'Call a subroutine'. This is a lot like an unconditional jump. But, it also uses a second stack – the subroutine stack - to remember where the subroutine was called. Then, when the subroutine itself has finished, the 'end a subroutine' call pops this value off the second stack, and changes the current position of the code execution, to this value.

The key thing when doing this is getting this value correct. Suppose your main loop is looking at character *i* in the program, and finds 'call a subroutine', followed by a label, followed by a newline. If *i* is pushed onto the subroutine stack, then the code will get stuck calling that subroutine: when the subroutine calls 'end a subroutine', execution will return to point *i*, which duly calls the subroutine again. Instead, you must push the position in the code **immediately after the call a subroutine instruction**, i.e. after the newline that defines what label to jump to for the subroutine. (In any other language, this would denote the next line of code.)

(If `preprocessing` is true, these both do nothing.)

To test your code now, try the examples from http://compsoc.dur.ac.uk/whitespace/examples.php – they should all now work.

Now what?

If you have managed to write a fully functioning Whitespace interpreter, I'm impressed. Come to see me to receive a complimentary fist-bump and to show me your code.

If you want an idea of what to do next: write **a program that translates Whitespace into C++**, to then be compiled and executed using a C++ compiler. You can write this program using Java, but the translated program it produces has to be C++: Java doesn't support labels and the 'goto' command, which are necessary for implementing jumps and subroutines. If you actually want to do this, let me know, and I'll write a cheat-sheet so you can understand enough C++ to do this.