Homework 9: Building a Vic Simulator

In this project we build a Java implementation of the Vic computer described in lecture 7-1. The computer will operate on programs and inputs stored in text files. For simplicity, assume that the contents of these files – the programs and the data – are always valid. That is, the program file contains a sequence of valid instructions written in the *numeric version* of the Vic machine language (3-digit integers), and the input file contains a valid sequence of integers ranging from -999 to 999. The code that you are asked to write need not test the instructions and the inputs validity – it should simply execute and read them, respectively.

Requirements: In order to develop this project you must be familiar with the Vic computer architecture and its fetch-execute logic. The best way to get this knowledge is to write and execute a few programs in the Vic machine language. This can be done by playing with the Vic simulator available in http://www1.idc.ac.il/vic, and by doing Homework 6, if you haven't done it already.

Objectives: This homework entails two lessons. First, you have to implement a project that consists of several classes, which is a good exercise in object-oriented programming. As you will implement these classes, you will understand the logic behind the system architecture (the class structures). Hopefully, you will note that the architecture makes the system implementation simple and elegant. Second, upon completion of this project, you will understand, deeply, how computers work at the low level. This understanding will be hands-on, since you will build a computer yourself

Javadoc: The given skeletal Java files contain Javadoc documentation. You will learn the basics of Javadoc documentation later in the course. You will also learn how to use your IDE (like VS Code) to generate the APIs of the classes given in this project.

Registers

A *register* is a container that holds an integer. This is the basic storage unit of the Vic computer. In the Vic computer implementation described here, registers are implemented as objects constructed from the supplied `Register` class.

Implementation: See the given `Register` class skeleton and documentation. A `Register` object has one field: the register's current value, which is an `int`. Implement the `Register` class by completing the supplied class skeleton. Each method in this class can be implemented by one line of code.

Testing: Start with the supplied `Test` class. Make sure to test all the `Register` methods.

Note: the difference between `reg.getvalue()` and `reg.toString()` is that the former returns an `int` and the latter a `String`. When working with registers, you always need to work with their `int` values, except when you want to print these values.

Memory

"Memory" is an indexed sequence of registers. In the Vic computer implementation described here, the memory is represented as an object derived from the supplied `Memory` class.

Implementation: See the given `Memory` class skeleton and documentation. A `Memory` object has one field: an array of `Register` objects (more accurately, an array of *references* to `Register` objects). The class constructor gets the desired length of the memory as a parameter, and creates an array of so many registers. Implement the `Memory` class by completing its supplied class skeleton.

Testing: Start with the supplied `Test` class. Make sure to test all the `Memory` methods.

The `toString` method of the `Memory` class should generate a string that shows the first and last 10 memory registers. Suppose that `m` is an object of type `Memory`. Suppose you wish to display the contents of the memory, by writing `System.out.println(m);` Here is what you will typically get (the contents of the memory is just an example, of course):

```
0	100
1	200
2	-17
3	0
4	0
5	0
6	0
7	0
8	0
9	0

90	0
91	0
92	0
93	0
94	0
95	0
96	0
97	0
98	0
99	1
```

That's precisely the output format that your `toString` implementation should generate.

Tip: The `toString` method of the `Memory` class should create a single, long string that uses "`\t`" and "`\n`" formatting characters in order to produce the output shown above. The "`\t`" character creates an indentation (`t` = "tab"), and the "`\n`" character inserts a line break into the string (`n` = "newline"). In other words, the output that you see above is actually one long string, that includes formatting characters that create the appearance of separate lines etc.

Computer

The `Computer` class is the main module of this project. Start working on it only after you complete developing and testing the `Register` and `Memory` classes. That's a good example of unit-testing: When you start working on the Computer implementation, you will know that the computer's registers and memory are working correctly, and you can now concentrate on the rest of the project.

We will develop the `Computer` class in three stages, as follows.

Stage 1

Look at the supplied `Computer` class. Start by declaring the ten constants for representing the ten op-codes of the Vic machine language. The first declaration is supplied in the given code:

```
private final static int READ = 8;
```

From now on, constants like `READ` should represent the Vic op-codes throughout the code that you write (instead of codes like 8, which are unclear).

The next section of the supplied `Computer` code declares the fields of the `Computer` object: a *memory*, a *data register* (which is a register), and a *program counter* (also a register). The input and output units are considered external to the computer, and will be described later in this document.

Now write the class constructor, the `reset` method, and the `toString` method.

Testing: Write a test method in the `Test` class that creates a `Computer` object and prints its state (that's two lines of code...). You should get this output:

```
D register  = 0
PC register = 0
Memory state:
0     0
1     0
2     0
3     0
4     0
5     0
6     0
7     0
8     0
9     0

90    0
91    0
92    0
93    0
94    0
95    0
96    0
97    0
98    0
99    1
```

Tip: The first four lines of this output are created by the `toString` method of the `Computer` class. The method should create a string that uses "`\n`" characters for line breaks, as described before in this document.

Stage 2

We now turn to writing the `loadProgram` method, which loads a program into the computer's memory. This method reads the contents of a text file containing valid Vic instructions (strings consisting of 3-digit characters each), and stores the instructions in the computer's memory, starting at address 0. To do so, the method (i) sets the standard input to be `programFile`, and (ii) reads each line from standard input, and loads it into memory.

Testing: Write a test method in the `Test` class that creates a `Computer` object and loads a program into it. The test code will look something like this:

```
Computer vic = new Computer();
vic.loadProgram("program1.vic");
System.out.println(vic);
```

The output of this code should look like this:

```
D register  = 0
PC register = 0
Memory state:
0     399
1     900
2     399
3     900
4     0
5     0
6     0
7     0
8     0
9     0

90    0
91    0
92    0
93    0
94    0
95    0
96    0
97    0
98    0
99    1
```

Stage 3

We now turn to implement the `run` method – the workhorse of this program. Start by reading the method documentation. The method simulates the fetch-execute cycle of the computer. It starts by fetching the next instruction from memory, and parsing it into an op-code and an address, which are kept in local variables. If the op-code is `STOP`, the method prints the text "`Program terminated normally`" and terminates. Otherwise, the method executes the command, according to its op-code. There are 9 commands, and thus 9 execution possibilities. We propose implementing the execution of each command by a separate private method. For example, the following method implements the execution of the `LOAD` command:

```
private void execLoad (int addr) {
    dReg.setValue(m.getValue(addr));  // dReg = m[addr]
    pc.addOne();                      // PC++
}
```

(This code is based on the assumption that the data register, program counter, and memory fields are named `dReg`, `pc`, and `m`, respectively).

Note that the `execLoad` method advances the program counter. This is very important – otherwise in the next cycle (loop iteration) the `run` method will fail to fetch the correct instruction from the memory. Also note that some commands (like `LOAD`) simply increment the program counter, while other commands – the branching ones – require a more delicate handling of the program counter. The important thing to remember is that every one of the methods that implement the Vic commands must not only execute the command, but also update the program counter.

Testing: We recommend implementing and testing the commands in the following order:

- Implement and test the `LOAD` and `WRITE` commands: use `program1.vic`
- Implement and test the `ADD` command: use `program2.vic`
- Implement and test the `SUB` command: use `program3.vic`
- Implement and test the `STORE` command: use `program4.vic`

Start by writing the main loop of the `run` method. Next, for each of the commands listed above, implement its private method and test your work by running a test that looks like this:

```
Computer vic = new Computer();
vic.loadProgram("program1.txt");  // or some other program file
vic.run();
System.out.println(vic);
```

`Program1.vic` prints the number 1, twice. Therefore, the output of this test should look like this:

```
1
1
Program terminated normally
D register  = 1
PC register = 4
Memory state:
0     399
1     900
2     399
3     900
4     0
5     0
6     0
7     0
8     0
9     0

90    0
91    0
92    0
93    0
94    0
95    0
```

```
96    0
97    0
98    0
99    1
```

The first two lines in the output shown above – the two 1's – are the program's output. The text ""`Program terminated normally`" is printed by the `run` method (your implementation should print it also). The rest of the output is generated by the `System.out.println(vic)` statement.

Stage 4

We now turn to implement the `READ` command. In this implementation, the `input` of the computer is represented by `StdIn`. In order to read inputs, we initialize the standard input by associating it with a given text file that contains the program's inputs. This is done by the method `loadInput`, which is given.

Testing: To test the `loadInput` and `execRead` methods, use something like this test code:

```
Computer vic = new Computer();
vic.loadProgram("program5.vic");
vic.loadInput("input1.txt");
vic.run();
System.out.println(vic);
```

`Program5.vic` reads and writes a number, three times. The file `input1.txt` contains the numbers `10 20 30`. The output of this test should be as follows:

```
10
20
30
Run terminated normally
D register  = 30
PC register = 6
Memory state:
0     800
1     900
2     800
3     900
4     800
5     900
6     0
7     0
8     0
9     0

90    0
91    0
92    0
93    0
94    0
95    0
96    0
97    0
98    0
99    1
```

Stage 5

We now turn to implement Vic's branching commands: `GOTO`, `GOTOZ`, and `GOTOP`. As usual, each command should be implemented by a separate private method. We recommend doing it in two stages:

Implement and test the `GOTO` and `GOTOZ` commands: Use `program6.txt` and `input2.txt`. This Vic program uses a loop to read and write a sequence of numbers that ends with a `0`.

Implement and test the `GOTOP` command: Use `max2.vic` .and `input3.txt`. This Vic program reads two numbers, and writes the greater or equal of the two.

Stage 6

Mazel Tov! You've built a Vic computer, using Java. As a final test, use your computer to execute the Vic programs that you wrote in Homework 6. Here is how to do it.

Suppose that you wrote a program, say *myProg*, using Vic's symbolic / assembly language. In order to execute this program on your Java simulator, you must first translate it into Vic's numeric machine language. To do so, invoke the supplied Vic assembler from http://www1.idc.ac.il/vic, load the file *myProg*`.asm`, and translate it. Next, save the translated file by clicking the disk icon. As a technical convention, the resulting file is always named `asmcode.txt`. Therefore, to end the translation process, rename this file *myProg*`.vic.`

Next, create an input file (you can use the example inputs given in Homework 6), and name it, say, `myInput.txt`. Finally, run the program on your simulator, as follows:

```
% java VicSimulator myProg.vic myInput.txt
```

Submission

Before submitting your work for grading, make sure that your code is written according to our Java Coding Style Guidelines. In addition to the 3 code files, create one PDF document that lists the code of the classes that you wrote, with the indentation. You may want to experiment with several "paste special" options, for transferring the code from your code editor to your word processing software. Whatever you do, the printed code must be well indented, and easy to read a review. Use the font Consolas, or Arial, size 12. The name of this file should be `HW9Code.pdf`.

Compress the following files into a file named `HW9.zip`:

- `Register.java`
- `Memory.java`
- `Computer.java`

Upload two files to Moodle: `HW9.zip`, and `HW9Code.pdf`.

Submission deadline: December 19, 2021, 23:55.