Introduction to Computer Science, IDC Herzliya, 2016

HW 7: Building a Vic Simulator

In this project we build a software implementation of the Vic computer. The result is a simulator that can execute Vic program. The simulator will have the same functionality as the Vic simulator that you used in HW 6, minus the graphical user interface.

Before working on this project you must first familiarize yourself with the Vic computer, architecture, and programming. In short, it is assumed that you have done Homework 6.

Usage

`> java VicSimulator` *programFile inputFile*

Where *programFile* is a text file containing numeric 3-digit Vic commands, and *inputFile* is an optional text file containing numeric 3-digit inputs.

The simulator loads the commands from the program file into the simulated computer, and then executes the program. If the program is expected to process inputs, it reads these inputs from the input file. The simulator prints the program's outputs to standard output (by default, the screen).

To simplify matters, we assume that the program file contains a stream of valid Vic machine language commands, and that the input file contains a stream of valid integers ranging from -999 to 999. The Vic simulator does not test the validity of the commands and the inputs. It executes and reads them, as-is.

Implementation

The Vic simulator can be implemented in many different ways. This document describes an object-oriented implementation, using Java. In order to guide the implementation, we provide a proposed API. In other words, *we* play the role of the system architect, while *you* are the developer who has to take our design and make it work. We now turn to provide a bottom-up description of what has to be done, along with proposed implementation stages, unit-testing, and supplied test programs.

Registers

A *register* can be described as a container that holds an integer. This is the basic storage unit of the Vic computer. In this program, registers are implemented as instances (objects) of the `Register.java` class.

Implementation: a `Register` object has a single field: an `int` variable, holding the register's current value. Write the `Register.java` class, following its API.

Testing: write a testing method that creates two or three registers (e.g. `Register r1 = new Register()`). Next, proceed to test all the `Register` methods. For example, try sequences of commands like `System.out.println(r1.getValue()); r1.addOne(); System.out.println(r1.getValue());` Write and test similar command sequences for each one of the `Register` methods, printing relevant values before and after invoking the method.

Memory

Memory can be described as an indexed sequence of registers. In this program, the memory is implemented as an instance (object) of the `Memory.java` class. The Vic computer has only one memory space (sometimes referred to as the RAM), and therefore our program will use only one `Memory` object.

Implementation: 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 proceeds to create an array of so many registers. Write the `Memory` class according to its API.

Testing: write a tester (testing method) that creates a single `Memory` object and tests all the `Memory` methods. The test code should look something like this:

```
Memory m = new Memory(100);
m.setValue(0, 100);     // m[0] = 100
m.setValue(1, 200);     // m[1] = 200
m.setValue(2, -17);     // m[2] = -17
int v = m.getValue(1);  // v = m[1]
System.out.println(v);
System.out.println(m);  // Display the memory's state (first and last 10 registers)
// Etc.
```

The `toString` method of the `Memory` class should generate a string that shows the first and last 10 registers in the memory. Here is a typical output of the command `System.out.println(m)`:

```
 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    0
```

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

Computer

In this program, the Vic computer is implemented as an instance of the `Computer.java` class.

Stage 1

Start by declaring and initializing the three public constants described in the `Computer` API. Next, declare 10 constants for representing the 10 op-codes of the Vic machine language. For example, the first declaration can be: `private final static int READ = 8`; From now on, constants like `READ` (rather than the obscure literal "8") should be used to represent op-codes throughout your class code. Next, declare the four fields (private variables) that characterize a `Computer` object: a memory, a data register (which is a register), a program counter (which is also a register). The input will be read through the standard input, `StdIn`. More about this, later.

Now write the `Computer` class constructor, the `reset` method, and the `toString` method. Consult the `Computer` API for details.

Testing: write a tester that creates a `Computer` object and prints the computer's state (that's two lines of code...). The output should look like this:

```
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
```

Stage 2

We now turn to write the `loadProgram` method, which loads a program into the computer's memory. Recall that the simulator starts running via the command "`java ViSimulator` $programFile$", with an optional additional data file. We assume that $programFile$ contains a stream of valid Vic commands. With that in mind, the `loadProgram` method reads from $programFile$ and serializes its contents into the computer's memory, starting at address 0. In short, the method should (i) set the standard input to be $programFile$, and (ii) read each line from this file and load it into memory. Implementation tip: `stdIn`.

Testing: write a testing method that creates a `Computer` object and loads a program into it. The code will look something like this:

```
Computer vic = new Computer();  // or vic.reset(), if the computer already exists.
vic.loadProgram("program1.vic");
System.out.println(vic);
```

Assuming that the file `program1.vic` is in your working directory, 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 the Vic simulator program. Start by reading the `run` method API. The method simulates the computer's fetch-execute cycle by fetching the current command from memory, parsing the command into an op-code and an address, and storing the parsed values in variables (you may want to handle some of these actions using private methods). 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 can be used to implement the execution of the `LOAD` command:

```
// Assumes that the data register, program counter, and memory fields
// are named dReg, pc, and m, respectively.
private void execLoad (int addr) {
    dReg.setValue(m.getValue(addr));  // dReg = m[addr]
    pc.addOne();                      // pc++
}
```

Note that the `execLoad` method advances the program counter. This is very important; if the pc is not incremented properly, in the next cycle (simulated by the next loop iteration) the `run` method will fail to fetch the correct command from 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 the program counter must be handled as a side-effect of handling the respective commands.

Testing: we recommend implementing and testing the Vic commands in stages, and in the following order:

Implement and test the `LOAD` and `WRITE` commands; test your implementation using `program1.vic`.
Implement and test the `ADD` command; test your implementation using `program2.vic`.
Implement and test the `SUB` command; test your implementation using `program3.vic`.
Implement and test the `STORE` command; test your implementation using `program4.vic`.

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

```
Computer vic = new Computer();    // or vic.reset(), if the computer already exists
vic.loadProgram("program1.vic");  // or some other program file
vic.run();
System.out.println(vic);
```

The output should look like this:

```
1
1
Run 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 - the two 1's - are the output of the execution of the Vic program. Next, the `run` method prints the text `"Run terminated normally"`. The rest of the output is generated by the `System.out.println(vic)` command.

Stage 4

We now turn to implement the `READ` command. Recall that the input field of our simulated `Computer` is implemented using `StdIn`. In order to read inputs, we initialize the standard input by associating it with a given text file that contains the program's input. This is done by the method `loadInput`. So, go ahead and implement this method.

Next, implement the `READ` command. Basically, you need a private `execRead` method that gets the next line from the input and puts it in the data register (don't forget to advance the program counter).

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

```
Computer vic = new Computer();  // or vic.reset(), if the computer already exists.
vic.loadProgram("program5.vic");
vic.loadInput("input1.dat");
vic.run();
System.out.println(vic);
```

If your working directory includes the files `program5.vic` and `input1.dat`, this code should produce the following output:

```
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
```

The first three lines are the output of the Vic program that was executed.

Stage 5

We now turn to implement the `GOTO`, `GOTOZ`, and `GOTOP` commands. As usual, each command should be implemented by a separate private method. We recommend doing it in stages, as follows:

Implement and test the `GOTO` and `GOTOZ` commands; test your implementation using `program6.vic` and `input2.dat`.
Implement and test the `GOTOP` command; test your implementation using `max2.vic` (a program the prints the maximum of two given numbers) and `input3.dat` (a file containing two numbers).

Stage 6

Mazeltov! you are the proud owner of a Vic computer, built by you from the ground up. You can now test your computer by loading and running any Vic program that comes to your mind. If the program is written in the symbolic Vic language, you must first use the supplied Vic assembler to translate the program into a corresponding `*.vic` file. If the program is expected to operate on inputs, you must put the inputs in a `*.dat` input file. See the Usage section of this document for more details on executing Vic programs using your Vic simulator.

Documentation

External documentation: there is no need to write Javadoc, since the APIs of all the classes that you have to write are given.

Internal documentation: use your judgment to write comments that describe significant code segments or highlight anything that you think is worth highlighting.

Submission

Before submitting any program, take some time to inspect your code, and make sure that it is written according to our **Java Coding Style Guidelines**. Also, make sure that each program starts with the program header described in the **Homework Submission Guidelines**. Any deviations from these guidelines will result in points penalty.

Submit the following files only:

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

Deadline: Submit your assignment no later than Tuesday, January 10, 2017, 23:55.