**Introduction to Computer Science, IDC Herzliya, 2019**

# HW 8: 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 programs. The simulator will have the same functionality as the web-based Vic simulator that you used in HW 5, 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 mandatory text file of the form *fileName*`.vic`, containing numeric 3-digit Vic commands, and *inputFile* is an optional text file containing the program's inputs. Examples:

> `java VicSimulator print1To100.vic` (Prints the numbers 1 to 100, no input file)

> `java VicSimulator sum.vic sum.dat` (Prints the sum of the numbers in the input file)

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 output to standard output (by default, the screen).

Both the program file and the optional input file are assumed to contain integers between `-999` greater than `999`.

## Implementation

The Vic simulator can be implemented in 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 is a container that holds a value. 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 supplied API.

Testing: write a testing method that creates two or three registers. Next, test all the `Register` methods. For example, try something like this:

```
Register r1 = new Register(10)
System.out.println(r1);     // Should print 10
r1.addOne();
System.out.println(r1);     // Should print 11
Register r2 = new Register()
r2.setValue(r1.getValue())
System.out.println(r2);     // Should print 11
```

Write and test similar command sequences for each one of the `Register` constructors and methods, printing relevant values before and after invoking each method. Tip: Each `Register` method can be implemented with one line of code.

## Memory

The memory unit is an indexed sequence of registers. In this implementation, the memory is realized as an instance of the `Memory.java` class. The Vic computer has one memory space, sometimes referred to as the RAM. Therefore, our program will use one `Memory` object (one instance of the `Memory` class).

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, following its supplied 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);  // Displays the memory's state.
```

The `toString` method of the `Memory` class returns 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   1
```

That's the exact output that your program should generate.

## Computer

The Vic computer is implemented as an instance of the `Computer.java` class. We will use only one object of that 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 three fields (private variables) that characterize a `Computer` object: a *memory* device (which is a `Memory` object), a *data register* (which is a `Register` object), and a program counter (also a `Register` object).

Now write the `Computer` class constructor, the `reset` method, and the `toString` methods. 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 = 0
PC = 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. This method reads the data from *programFile* (see the *Usage Section* at the beginning of this document), line by line, and loads the data – a stream of 3-digit numbers – into the computer's memory, starting at address 0. Implementation tip: Use the services of the `stdIn` class, both for initializing the program file, and for reading from it.

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 `program1.vic` file is in your working directory, the output of this code should look like this:

```
D = 0
PC = 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. 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 `"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 that lent 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 using client code 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);
```

If your test code has loaded `program1.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 loaded Vic program (`program1.vic`). Next, the `run` method prints `"Run terminated normally"`. The rest of the output is generated by the `System.out.println(vic)` command.

**Stage 4**

Now turn to implement the `READ` command. Before we read any inputs, we must inform the computer where the inputs are coming from. This is done by the `Computer`’s `setInput` method.

Going back to the *Usage Section* at the beginning of this document, note that the inputs are coming from a file whose name is the second (optional) command-line argument. Hint: Implement the `setInput` method using `StdIn`. The implementation is one line of code.

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

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

```
      Computer vic = new Computer();  // or vic.reset(), if the computer already exists.
      vic.loadProgram("program5.vic");
      vic.setInput("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 = 30
PC = 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**

Now turn to implementing 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 that prints the maximum of two given numbers) and `input3.dat` (a file containing two numbers).

## VicSimulator

This class is the driver of the Vic Simulator. It consists of a single method, named `main`. The `main` method creates a `Computer` object (we suggest naming it `vic`), and loads into it the code from the supplied *programFile* (first command-line argument). This can done by calling the `loadProgram` method on the `vic` object. If there is a second command-line argument, `main` calls the `setInput` method on `vic`. Finally, `main` calls the `run` method, and then prints the state of the `vic` computer.

Mazel Tov! 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 Vic Assembler (available in www.idc.ac.il/vic) to translate it into a corresponding `*.vic` file. If the program is expected to read one or more inputs, you must put the inputs in a `*.dat` input file. See the *Usage Section* at the beginning 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`
- `VicSimulator.java`

**Deadline:** Submit your assignment no later than **19/01/2020, 23:55**