

```

1 /**
2  * Represents a register.
3  * A register is the basic storage unit of the Vic computer.
4  */
5
6 public class Register {
7
8     private int value; // the current value of this register
9
10    /** Constructs a register and sets its value to 0. */
11    public Register() {
12        this.setValue(0);
13    }
14
15    public Register(int val) {
16        this.setValue(val);
17    }
18
19    /**
20     * Sets the value of this register.
21     *
22     * @param v the value to which the register will be set.
23     */
24    public void setValue(int val) {
25        this.value = val;
26    }
27
28    /** Increments the value of this register by 1. */
29    public void addOne() {
30        this.value = this.value + 1;
31    }
32
33    /**
34     * Returns the value of this register.
35     *
36     * @return the current value of this register, as an int.
37     */
38    public int getValue() {
39        return this.value;
40    }
41
42    /**
43     * Returns a textual representation of the value of this register.
44     *
45     * @return Returns the value of this register, as a String.
46     */
47    public String toString() {
48        return (" " + this.value);
49    }
50 }

```

```

1
2 /**
3  * Represents a random access memory (RAM) unit. A RAM is an indexed sequence of
4  * registers
5  * that enables reading from, or writing to, any individual register according
6  * to a given index.
7  * The index is typically called "address". The addresses run from 0 to the
8  * memory's size, minus 1.
9  */
10
11 public class Memory {
12
13     private Register[] m; // an array of Register objects
14
15     /**
16      * Constructs a memory of size registers, and sets all the register values to 0.
17      * Each register in the memory is a Register object.
18      *
19      * @param size the size (number of registers) of this memory.
20      */
21     public Memory(int size) {
22         this.m = new Register[size];
23         for (int i = 0; i < m.length; i++) {
24             this.m[i] = new Register();
25         }
26     }
27
28     /** Sets the values of all the registers in this memory to 0. */
29     public void reset() {
30         for (int i = 0; i < this.m.length; i++) {
31             m[i].setValue(0);
32         }
33     }
34
35     /**
36      * Returns the value of the register whose address is the given address.
37      *
38      * @param address the address of the register.
39      * @return the value of the register, as an int.
40      */
41     public int getValue(int address) {
42         return (this.m[address].getValue());
43     }
44
45     /**
46      * Sets the register in the given address to the given value.
47      *
48      * @param address the address of the register.
49      * @param value the register's value will be set to value.
50      */
51     public void setValue(int address, int value) {
52         this.m[address].setValue(value);
53     }
54
55     /**
56      * Returns the memory's contents, as a formatted string. To avoid clutter,
57      * returns only the
58      * first 10 registers (where the top of the program normally resides) and the
59      * last 10 registers
60      * (where the variables normally reside). For each register, returns the
61      * register's address and
62      * value.
63      */

```

```
64 public String toString() {
65     String text = "";
66     for (int i = 0; i < 10; i++) {
67         text += (i + "\t" + this.m[i].toString() + "\n");
68     }
69     text += "\n";
70     for (int j = this.m.length - 10; j < this.m.length; j++) {
71         text += j + "\t" + this.m[j].toString() + "\n";
72     }
73     return text;
74 }
75 }
```

```

1
2 /**
3  * Represents a Vic computer.
4  * It is assumed that users of this class are familiar with the Vic computer,
5  * and the Vic machine language, described in www1.idc.ac.il/vic.
6  * <br/>
7  * The Computer's hardware consists of the following components:
8  * <UL>
9  * <LI>Data register: a register.
10 * <LI>Program counter: a register.
11 * <LI>Input unit: a stream of numbers. In this implementation, the input unit
12 * is simulated
13 * by a text file. When the computer is instructed to execute a READ
14 * instruction, it reads
15 * the next number from this file and puts it in the data register.
16 * <LI>Output unit: a stream of numbers. In this implementation, the output unit
17 * is simulated by
18 * standard output (by default, the console).
19 * When the computer is instructed to execute a WRITE instruction, it writes the
20 * current
21 * value of the data register to the standard output.
22 * <LI>Processor: In this implementation, the processor is emulated by the run
23 * method of this class.
24 * </UL>
25 * The Computer executes programs written in the numeric Vic machine language.
26 * The program is stored in a text file that can be loaded into the computer's
27 * memory.
28 * This is done by the loadProgram method of this class.
29 */
30
31 public class Computer {
32
33     /**
34      * This constant represents the size of the memory unit of this Computer
35      * (number of memory registers).
36      */
37     public final static int MEM_SIZE = 100;
38
39     /**
40      * This constant represents the memory address at which the constant 0 is
41      * stored.
42      */
43     public final static int LOCATION_OF_ZERO = MEM_SIZE - 2;
44
45     /**
46      * This constant represents the memory address at which the number 1 is stored.
47      */
48     public final static int LOCATION_OF_ONE = MEM_SIZE - 1;
49
50     // Op-code definitions:
51     private final static int ADD = 1;
52     private final static int SUB = 2;
53     private final static int LOAD = 3;
54     private final static int STORE = 4;
55     private final static int GOTO = 5;
56     private final static int GOTOZ = 6;
57     private final static int GOTOP = 7;
58     private final static int READ = 8;
59     private final static int WRITE = 9;
60     private final static int STOP = 0;
61
62     /** The Computer consists of a Memory unit, and two registers, as follows: */
63     private Memory m;

```

```

64 private Register dReg;
65 private Register pc;
66
67 /**
68  * Constructs a Vic computer. Specifically:
69  * Constructs a memory that has MEM_SIZE registers, a data register,
70  * and a program counter. Next, resets the computer (see the reset method API).
71  *
72  * Note: the initialization of the input unit and the loading of a program into
73  * memory are not done by the constructor. This is done by the public methods
74  * loadInput and loadProgram, respectively.
75  */
76 public Computer() {
77     this.m = new Memory(MEM_SIZE);
78     this.dReg = new Register();
79     this.pc = new Register();
80     reset();
81 }
82
83 /**
84  * Resets the computer. Specifically:
85  * Resets the memory, sets the memory registers at addresses LOCATION_OF_ZERO
86  * and LOCATION_OF_ONE to 0 and to 1, respectively, sets the data register
87  * and the program counter to 0.
88  */
89 public void reset() {
90     this.m.reset();
91     this.m.setValue(LOCATION_OF_ONE, 1);
92     this.m.setValue(LOCATION_OF_ZERO, 0);
93     this.dReg.setValue(0);
94     this.pc.setValue(0);
95 }
96
97 /**
98  * Executes the program currently stored in memory.
99  * This is done by affecting the following fetch-execute cycle:
100  * Fetches from memory the next instruction (3-digit number), i.e. the contents
101  * of the
102  * memory register whose address is the current value of the program counter.
103  * Extracts from this word the op-code (left-most digit) and the address (next 2
104  * digits).
105  * Next, executes the command specified by the op-code, using the address if
106  * necessary.
107  * As a side-effect of executing the instruction, modifies the program counter.
108  * Next, loops to fetch the next instruction, and so on.
109  */
110 public void run() {
111     if (this.m.getValue(pc.getValue()) / 100 == STOP) {
112         execSTOP();
113     }
114     if (this.m.getValue(pc.getValue()) / 100 == ADD) {
115         int addr = m.getValue(pc.getValue()) % 100;
116         execADD(addr);
117     } else if (this.m.getValue(pc.getValue()) / 100 == SUB) {
118         int addr = m.getValue(pc.getValue()) % 100;
119         execSUB(addr);
120     } else if (this.m.getValue(pc.getValue()) / 100 == LOAD) {
121         int addr = m.getValue(pc.getValue()) % 100;
122         execLoad(addr);
123     } else if (this.m.getValue(pc.getValue()) / 100 == STORE) {
124         int addr = m.getValue(pc.getValue()) % 100;
125         execSTORE(addr);
126     } else if (this.m.getValue(pc.getValue()) / 100 == GOTO) {

```

```

127         int addr = m.getValue(pc.getValue()) % 100;
128         execGOTO(addr);
129     } else if (this.m.getValue(pc.getValue()) / 100 == GOTOZ) {
130         int addr = m.getValue(pc.getValue()) % 100;
131         execGOTOZ(addr);
132     } else if (this.m.getValue(pc.getValue()) / 100 == GOTOY) {
133         int addr = m.getValue(pc.getValue()) % 100;
134         execGOTOY(addr);
135     } else if (this.m.getValue(pc.getValue()) / 100 == READ) {
136         execREAD();
137     } else if (this.m.getValue(pc.getValue()) / 100 == WRITE) {
138         execWRITE();
139     }
140 }
141
142 // Private execution routines, one for each Vic command
143 private void execADD(int addr) {
144     dReg.setValue(dReg.getValue() + m.getValue(addr));
145     pc.addOne();
146     run();
147 }
148
149 private void execSUB(int addr) {
150     dReg.setValue(dReg.getValue() - m.getValue(addr));
151     pc.addOne();
152     run();
153 }
154
155 private void execLoad(int addr) {
156     dReg.setValue(m.getValue(addr));
157     pc.addOne();
158     run();
159 }
160
161 private void execSTORE(int addr) {
162     m.setValue(addr, dReg.getValue());
163     pc.addOne();
164     run();
165 }
166
167 private void execGOTO(int addr) {
168     pc.setValue(addr);
169     run();
170 }
171
172 private void execGOTOZ(int addr) {
173     if (dReg.getValue() == 0) {
174         pc.setValue(addr);
175     } else {
176         pc.addOne();
177     }
178     run();
179 }
180
181 private void execGOTOP(int addr) {
182     if (dReg.getValue() > 0) {
183         pc.setValue(addr);
184     } else {
185         pc.addOne();
186     }
187     run();
188 }
189

```

```

190 private void execREAD() {
191     dReg.setValue(StdIn.readInt());
192     pc.addOne();
193     run();
194 }
195
196 private void execWRITE() {
197     System.out.println(dReg.getValue());
198     pc.addOne();
199     run();
200 }
201
202 private void execSTOP() {
203     System.out.println("Program terminated normally");
204 }
205
206 // Implement the other private methods here (execRead, execWrite, execAdd,
207 // etc.).
208 // For each mehod, you have to write its siganture, and implement it.
209
210 /**
211  * Loads a program into memory, starting at address 0, using the standard input.
212  * The program is stored in a text file whose name is the given fileName.
213  * It is assumed that the file contains a stream of valid commands written
214  * in the numeric Vic machine language (described in www1.idc.ac.il/vic).
215  * The program is stored in the memory, starting at address 0.
216  */
217 public void loadProgram(String fileName) {
218     int index = 0;
219     StdIn.setInput(fileName);
220     while (!StdIn.isEmpty()) {
221         this.m.setValue(index, StdIn.readInt());
222         index++;
223     }
224 }
225
226 /**
227  * Initializes the input unit from a given text file using the standard input.
228  * It is assumed that the file contains a stream of valid data values,
229  * each being an integer in the range -999 to 999.
230  * Each time the computer is instructed to execute a READ instruction,
231  * the next line from this file is read and placed in the data register
232  * (this READ logic is part of the run method implementation).
233  * The role of this method is to initialize the file in order to
234  * enable the execution of subsequent READ commands.
235  */
236 public void loadInput(String fileName) {
237     StdIn.setInput(fileName);
238 }
239
240 /**
241  * This method is used for debugging purposes.
242  * It displays the current contents of the data register,
243  * the program counter, and the first and last 10 memory cells.
244  */
245 public String toString() {
246     return ("D register = " + dReg + "\nPC register = " + pc + "\nMemory state:\n" +
m.toString());
247 }
248 }

```