```java

/**
 * Represents a random access memory (RAM) unit. A RAM is an indexed sequence of
 * registers
 * that enables reading from, or writing to, any individual register according
 * to a given index.
 * The index is typically called "address". The addresses run from 0 to the
 * memory's size, minus 1.
 */

public class Memory {

    private Register[] m; // an array of Register objects

    /**
     * Constructs a memory of size registers, and sets all the register values to 0.
     * Each register in the memory is a Register object.
     *
     * @param size the size (number of registers) of this memory.
     */
    public Memory(int size) {
        this.m = new Register[size];
        for (int i = 0; i < m.length; i++) {
            this.m[i] = new Register();
        }
    }

    /** Sets the values of all the registers in this memory to 0. */
    public void reset() {
        for (int i = 0; i < this.m.length; i++) {
            m[i].setValue(0);
        }
    }

    /**
     * Returns the value of the register whose address is the given address.
     *
     * @param address the address of the register.
     * @return the value of the register, as an int.
     */
    public int getValue(int address) {
        return (this.m[address].getValue());
    }

    /**
     * Sets the register in the given address to the given value.
     *
     * @param address the address of the register.
     * @param value   the register's value will be set to value.
     */
    public void setValue(int address, int value) {
        this.m[address].setValue(value);
    }

    /**
     * Returns the memory's contents, as a formated string. To avoid clutter,
     * returns only the
     * first 10 registers (where the top of the program normally resides) and the
     * last 10 registers
     * (where the variables normally reside). For each register, returns the
     * register's address and
     * value.
     */
    public String toString() {
        String text = "";
        for (int i = 0; i < 10; i++) {
            text += (i + "\t" + this.m[i].toString() + "\n");
        }
        text += "\n";
        for (int j = this.m.length - 10; j < this.m.length; j++) {
            text += j + "\t" + this.m[j].toString() + "\n";
        }
        return text;
    }
}
```

```java
/**
 * Represents a register.
 * A register is the basic storage unit of the Vic computer.
 */

public class Register {

    private int value; // the current value of this register

    /** Constructs a register and sets its value to 0. */
    public Register() {
        this.setValue(0);
    }

    public Register(int val) {
        this.setValue(val);
    }

    /**
     * Sets the value of this register.
     *
     * @param v the value to which the register will be set.
     */
    public void setValue(int val) {
        this.value = val;
    }

    /** Increments the value of this register by 1. */
    public void addOne() {
        this.value = this.value + 1;
    }

    /**
     * Returns the value of this register.
     *
     * @return the current value of this register, as an int.
     */
    public int getValue() {
        return this.value;
    }

    /**
     * Returns a textual representation of the value of this register.
     *
     * @return Returns the value of this register, as a String.
     */
    public String toString() {
        return ("" + this.value);
    }
}
```

```java

/**
 * Represents a Vic computer.
 * It is assumed that users of this class are familiar with the Vic computer,
 * and the Vic machine language, described in www1.idc.ac.il/vic.
 * <br/>
 * The Computer's hardware consists of the following components:
 * <UL>
 * <LI>Data register: a register.
 * <LI>Program counter: a register.
 * <LI>Input unit: a stream of numbers. In this implementation, the input unit
 * is simulated
 * by a text file. When the computer is instructed to execute a READ
 * instruction, it reads
 * the next number from this file and puts it in the data register.
 * <LI>Output unit: a stream of numbers. In this implementation, the output unit
 * is simulated by
 * standard output (by default, the console).
 * When the computer is instructed to execute a WRITE instruction, it writes the
 * current
 * value of the data register to the standard output.
 * <LI>Processor: In this implementation, the processor is emulated by the run
 * method of this class.
 * </UL>
 * The Computer executes programs written in the numeric Vic machine language.
 * The program is stored in a text file that can be loaded into the computer's
 * memory.
 * This is done by the loadProgram method of this class.
 */

public class Computer {

    /**
     * This constant represents the size of the memory unit of this Computer
     * (number of memory registers).
     */
    public final static int MEM_SIZE = 100;

    /**
     * This constant represents the memory address at which the constant 0 is
     * stored.
     */
    public final static int LOCATION_OF_ZERO = MEM_SIZE - 2;

    /**
     * This constant represents the memory address at which the number 1 is stored.
     */
    public final static int LOCATION_OF_ONE = MEM_SIZE - 1;

    // Op-code definitions:
    private final static int ADD = 1;
    private final static int SUB = 2;
    private final static int LOAD = 3;
    private final static int STORE = 4;
    private final static int GOTO = 5;
    private final static int GOTOZ = 6;
    private final static int GOTOP = 7;
    private final static int READ = 8;
    private final static int WRITE = 9;
    private final static int STOP = 0;

    /** The Computer consists of a Memory unit, and two registers, as follows: */
    private Memory m;
    private Register dReg;
    private Register pc;

    /**
     * Constructs a Vic computer. Specifically:
     * Constructs a memory that has MEM_SIZE registers, a data register,
     * and a program counter. Next, resets the computer (see the reset method API).
     *
     * Note: the initialization of the input unit and the loading of a program into
     * memory are not done by the constructor. This is done by the public methods
     * loadInput and loadProgram, respectively.
     */
    public Computer() {
        this.m = new Memory(MEM_SIZE);
        this.dReg = new Register();
        this.pc = new Register();
        reset();
    }

    /**
     * Resets the computer. Specifically:
     * Resets the memory, sets the memory registers at addresses LOCATION_OF_ZERO
     * and LOCATION_OF_ONE to 0 and to 1, respectively, sets the data register
     * and the program counter to 0.
     */
    public void reset() {
        this.m.reset();
        this.m.setValue(LOCATION_OF_ONE, 1);
        this.m.setValue(LOCATION_OF_ZERO, 0);
        this.dReg.setValue(0);
        this.pc.setValue(0);
    }

    /**
     * Executes the program currently stored in memory.
     * This is done by affecting the following fetch-execute cycle:
     * Fetches from memory the next instruction (3-digit number), i.e. the contents
     * of the
     * memory register whose address is the current value of the program counter.
     * Extracts from this word the op-code (left-most digit) and the address (next 2
     * digits).
     * Next, executes the command specified by the op-code, using the address if
     * necessary.
     * As a side-effect of executing the instruction, modifies the program counter.
     * Next, loops to fetch the next instruction, and so on.
     */
    public void run() {
        if (this.m.getValue(pc.getValue()) / 100 == STOP) {
            execSTOP();
        }
        if (this.m.getValue(pc.getValue()) / 100 == ADD) {
            int addr = m.getValue(pc.getValue()) % 100;
            execADD(addr);
        } else if (this.m.getValue(pc.getValue()) / 100 == SUB) {
            int addr = m.getValue(pc.getValue()) % 100;
            execSUB(addr);
        } else if (this.m.getValue(pc.getValue()) / 100 == LOAD) {
            int addr = m.getValue(pc.getValue()) % 100;
            execLoad(addr);
        } else if (this.m.getValue(pc.getValue()) / 100 == STORE) {
            int addr = m.getValue(pc.getValue()) % 100;
            execSTORE(addr);
        } else if (this.m.getValue(pc.getValue()) / 100 == GOTO) {
            int addr = m.getValue(pc.getValue()) % 100;
            execGOTO(addr);
        } else if (this.m.getValue(pc.getValue()) / 100 == GOTOZ) {
            int addr = m.getValue(pc.getValue()) % 100;
            execGOTOZ(addr);
        } else if (this.m.getValue(pc.getValue()) / 100 == GOTOP) {
            int addr = m.getValue(pc.getValue()) % 100;
            execGOTOP(addr);
        } else if (this.m.getValue(pc.getValue()) / 100 == READ) {
            execREAD();
        } else if (this.m.getValue(pc.getValue()) / 100 == WRITE) {
            execWRITE();
        }
    }

    // Private execution routines, one for each Vic command
    private void execADD(int addr) {
        dReg.setValue(dReg.getValue() + m.getValue(addr));
        pc.addOne();
        run();
    }

    private void execSUB(int addr) {
        dReg.setValue(dReg.getValue() - m.getValue(addr));
        pc.addOne();
        run();
    }

    private void execLoad(int addr) {
        dReg.setValue(m.getValue(addr));
        pc.addOne();
        run();
    }

    private void execSTORE(int addr) {
        m.setValue(addr, dReg.getValue());
        pc.addOne();
        run();
    }

    private void execGOTO(int addr) {
        pc.setValue(addr);
        run();
    }

    private void execGOTOZ(int addr) {
        if (dReg.getValue() == 0) {
            pc.setValue(addr);
        } else {
            pc.addOne();
        }
        run();
    }

    private void execGOTOP(int addr) {
        if (dReg.getValue() > 0) {
            pc.setValue(addr);
        } else {
            pc.addOne();
        }
        run();
    }

```

```java
    private void execREAD() {
        dReg.setValue(StdIn.readInt());
        pc.addOne();
        run();
    }

    private void execWRITE() {
        System.out.println(dReg.getValue());
        pc.addOne();
        run();
    }

    private void execSTOP() {
        System.out.println("Program terminated normally");
        pc.addOne();
    }

    // Implement the other private methods here (execRead, execWrite, execAdd,
    // etc.).
    // For each mehod, you have to write its siganture, and implement it.

    /**
     * Loads a program into memory, starting at address 0, using the standard input.
     * The program is stored in a text file whose name is the given fileName.
     * It is assumed that the file contains a stream of valid commands written
     * in the numeric Vic machine language (described in www1.idc.ac.il/vic).
     * The program is stored in the memory, starting at address 0.
     */
    public void loadProgram(String fileName) {
        int index = 0;
        StdIn.setInput(fileName);
        while (!StdIn.isEmpty()) {
            this.m.setValue(index, StdIn.readInt());
            index++;
        }
    }

    /**
     * Initializes the input unit from a given text file using the standard input.
     * It is assumed that the file contains a stream of valid data values,
     * each being an integer in the range -999 to 999.
     * Each time the computer is instructed to execute a READ instruction,
     * the next line from this file is read and placed in the data register
     * (this READ logic is part of the run method implementation).
     * The role of this method is to initialize the file in order to
     * enable the execution of subsequent READ commands.
     */
    public void loadInput(String fileName) {
        StdIn.setInput(fileName);
    }

    /**
     * This method is used for debugging purposes.
     * It displays the current contents of the data register,
     * the program counter, and the first and last 10 memory cells.
     */
    public String toString() {
        return ("D register  = " + dReg + "\nPC register = " + pc + "\nMemory state:\n" +
m.toString());
    }
}
```