LA3
Instruction Execution Simulator

Due Date
Friday 10/18/19 @ 11:59pm

Objectives

- Review Inheritance
- Review Polymorphism

Background

Any high level program language will finally be compiled into machine-readable instructions and executed on a machine with a CPU and memory. In this assignment you are asked to implement a Java application to simulate the execution of a list of instructions on a CPU and on locations in memory. The components of the application are as follows:

- **CPU**
  - CPU has two registers: "**ax**" and "**bx**" for arithmetic operations. Each register can store an integer.
- **Memory**
  - Each unit in memory can store an integer. The memory can store 10 integers. The integers are stored in an array. The address of each memory unit is its index in the array.
- **Instruction**
  - This is the superclass (an **abstract** class) of the different types of instruction classes. Each instruction contains one operation code (opcode) followed by two arguments. The opcode specifies the type of the instruction which will be executed on the values stored in the locations indicated by the two arguments. There are 4 types of instructions in total:
  - ➢ **Load** Instruction

    The execution of a **Load** Instruction will load an integer from the memory to the register. The first argument (after the opcode) specifies the register name (ax or bx). The second argument is the memory address. For example, "load,ax,1" means load the integer from memory address 1 to the "ax" register.
  - ➢ **Store** Instruction

    The execution of a **Store** Instruction will store the integer value in a register to memory. The first argument specifies the register name. The second argument is the memory address at which to store the register value. For example, "store,ax,3" means store the value obtained from the "ax" register in memory address 3.
  - ➢ **Add** Instruction

    The execution of an **Add** Instruction will perform an add operations on two integers. The first argument is the register name in which the first integer is stored. The second argument can be either a register name or an integer which indicates the memory address of the second integer. The result of the **add** operation will be stored at the register specified by the first argument. For example, “add,ax,1” will store the value “ax+1” in the ax register and “add,ax,bx” will store the value “ax+bx” in the ax register.

➢ **Sub** Instruction (for subtraction)

The execution of a **Sub** Instruction will perform a subtraction operation on two integers. The first argument is the register name in which the first integer is stored. The second operand can be either a register name or an integer which indicates the source of the second integer. The result of the subtraction operation will be stored at the register specified by the first argument. For example, “sub,ax,1” will store the value “ax-1” in the ax register and “sub,ax,bx” will store the value “ax-bx” in the ax register.

Problem Specification

Your application should load instructions from the input file provided (named “**instructions.txt**”. Initially, the value of the CPU registers are zero, and each location in the memory unit (which is represented by a one-dimensional array) is set to its index (i.e. memory[0] = 0; memory[1] = 1; etc. . Your program should execute all the instructions in the input file and then print out the status of the memory and CPU (whose values may be updated as the program executes).

Each instruction has the following format:

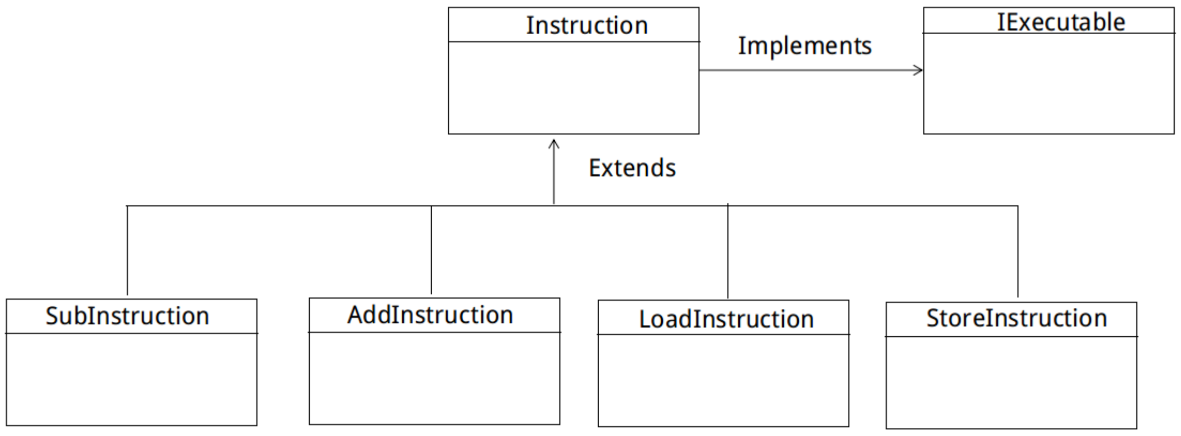
opcode,arg1,arg2

The first line in the input file (instructions.txt) is an integer specifying the number of instructions in the file.

The three components of an instruction are separated by commas.

Design Requirements

Your program structure should follow the UML diagram below.



Here is a list classes you need to write:

- **CPU**

  - This class implements the ICPU interface and contains the registers used for executing the instructions.
- **Memory**
  - This implements the IMemory interface and is implemented using an array.
- **Instruction**
  - This implements the IExecutable interface. It is an **abstract** class and serves as a superclass for the classes of the different types of instructions. Since the “execute(…)” method defined in the IExecutable interface will vary from one type of instruction to another, it is executed by each of the subclasses of class Instruction (these subclasses are specified below). Note that **interface IExecutable has a static method which must be implemented in the interface**. Since this method is static, it can be called just by prefixing its name with the interface name followed by a period.
- **AddInstruction**
  - Extends class Instruction and overrides the interface method “execute(…)”.
- **SubInstruction**
  - Extends class Instruction and overrides the interface method “execute(…)”.
- **LoadInstruction**
  - Extends class Instruction and overrides the interface method “execute(…)”.
- **StoreInstruction**
  - Extends class Instruction and overrides the interface method “execute(…)”.

An example of the input and expected output is given below. **Your program output must follow the format given in the example output below and display the same content.** Test you program with the test case provided in the Dropbox.

Example Input:

```
add,ax,1
store,ax,1
add,bx,2
store,bx,2
load,ax,1
load,bx,2
add,ax,bx
store,ax,3
add,ax,6
store,ax,8
sub,ax,3
store,ax,9
```

Example Output:

```
Initial memory contents:
Address 0: 0
Address 1: 1
Address 2: 2
Address 3: 3
Address 4: 4
Address 5: 5
Address 6: 6
Address 7: 7
Address 8: 8
Address 9: 9

Program output:

Memory Status:
Address 0: 0
Address 1: 1
Address 2: 2
Address 3: 3
Address 4: 4
Address 5: 5
Address 6: 6
Address 7: 7
Address 8: 9
Address 9: 6

CPU Status:
ax:6 bx:2
```

The interfaces to be implemented (as specified above) are provided below.

```
public interface IExecutable {
	/**
	 * Overridden by the four subclasses of abstract class Instruction.
	 *
	 * It checks to see which register(s) and memory locations (if
applicable)
	 * are to be used for this instruction and sets the relevant register or
	 * memory location with the result / value based on the specific
instruction
	 * (add, subtract, load or store).
	 *
	 * @param cpu      The CPU object with the registers ax, bx
	 * @param memory   The memory object to be used for retrieving or
storing data
	 */
	void execute(ICPU cpu, IMemory memory);

	/**
	 * This is a static method and so must be implemented here. YOU WILL
NEED
	 * TO WRITE THE IMPLEMENTATION FOR THIS METHOD.
	 *
	 * It reads the data in the input file and stores the individual
instructions
	 * in an array of IExecutable objects. In order to determine what kind
of
	 * instruction to instantiate, it needs to check the first token (or
word)
	 * on each line and then create the corresponding Instruction object
	 * (passing the parameters for that object to the relevant constructor
in the process).
	 *
	 * Remember that the first line in the input file specifies the number 
of 
	 * instructions stored in it.
	 * 
	 * @param file	The File object referencing the input file.
	 * @return	The array of IExecutable objects / instructions.
	 * @throws IOException	In case the input file is not found.
	 */
	static IExecutable[] loadInstructionsFromFile(File file) throws 
IOException{

	// INCLUDE THE CODE FOR THIS STATIC METHOD.

	}
	
	/**
	 * A getter for the first argument in an instruction.
	 * 
	 * @return	The first argument.
	 */
	String getArg1();
	
	/**
	 * A setter for the first argument.
	 * 
	 * @param arg1	The first argument.
	 */
	void setArg1(String arg1);
	
	/**
	 * A getter for the second argument in an instruction.
	 * 
	 * @return	The second argument.
	 */
	String getArg2();
	
	/**
	 * A setter for the second argument.
	 * 
	 * @param arg2	The second argument.
	 */
	void setArg2(String arg2);
	
	/**
	 * A getter for the operation code that specifies the type of operation
	 * or instruction to be performed.
	 * 
	 * @return	The type of operation or instruction.
	 */
	String getOpcode();
	
	/**
	 * A setter for the type of operation to be performed.
	 * 
	 * @param opcode	The operation to be performed.
	 */
	void setOpcode(String opcode);
```

```
}	// End of IExecutable


public interface ICPU {
	/**
	 * Prints out the status of the 2 CPU registers (ax and bx) -
	 * i.e the values stored in the registers.
	 */
	void printCPUStatus();

	/**
	 * Returns the value stored in register ax.
	 *
	 * @return  Value in ax.
	 */
	int getAx();

	/**
	 * Updates the value in ax with the value in the parameter.
	 *
	 * @param axNew value for ax.
	 */
	void setAx(int ax);

	/**
	 * Returns the value stored in register bx.
	 *
	 * @return
	 */
	int getBx();

	/**
	 * Updates the value in bx with the value in the parameter.
	 *
	 * @param bx
	 */
	void setBx(int bx);

}	// End of ICPU


public interface IMemory {

	// NOTE: You should have a constructor for your Memory class. This
	// constructor should initialize each location in the memory array
	// with its index in the array: so memory[0] = 0; memory[1] = 1; etc.
	// The constructor should also print out the initial contents of the
	// memory.

	/**
	 * Prints out the status of the memory (a one-dimensional array)
	 * (status => the contents of the array representing the memory).
	 */
	void printMemoryStatus();

	/**
```

```
         * Return the memory array.
         *
         * @return  The memory array.
         */
        int[] getMemoryContent();

        /**
         * Update the memory array with the one referenced in the parameter.
         *
         * @param memoryContent The array with which to update the memory.
         */
        void setMemoryContent(int[] memoryContent);

}
```

You are required to use the test class provided below to test your program. Do **NOT** make any changes to the test class or to any of the interfaces provided.

```
public class Main {

        public static void main(String[] args) throws IOException {
                File file = new File("instructions.txt");

                IExecutable[] instructions =
IExecutable.loadInstructionsFromFile(file);
                ICPU cpu = new CPU();
                IMemory memory = new Memory();

                for(IExecutable ins:instructions) {
                        ins.execute(cpu, memory);
                }
                memory.printMemoryStatus();
                cpu.printCPUStatus();
        }
}
```

Implementation Phase

Using the pseudocode developed, write the Java code for your assignment. This is a two-week assignment.

Testing Phase

- You can verify that your program is giving the correct output by comparing your output with a walk-through done by hand.
- You may also generate additional input files similar to the one provided to test your program further.

Additional Requirements

A proper *design* (with detailed pseudocode) and proper *testing* are essential.

*Note: **Correct pseudocode development** will be worth **40%** of the total LA grade.*

You will need to **generate Javadoc** for your project. Follow the steps shown in class (a copy of these steps can be found on the Content page in Elearning. If you follow the steps accurately, a "**doc**" folder (for Javadoc) should be created in your project.

Coding Standards

You must adhere to all conventions in the CS 1120 Java coding standards (available on Elearning for your Lab). This includes the use of white spaces and indentations for readability, and the use of comments to explain the meaning of various methods and attributes. Be sure to follow the conventions also for naming classes, variables, method parameters and methods.

Assignment Submission

- Generate a .zip file that contains all your files including:
  - Program Files
  - Any input or output files
  - Detailed pseudocode for your program

  You can refer to the link provided for instructions on how to export your project to a .zip file ("How to submit a programming assignment").

- Submit the .zip file to the appropriate folder on ELearning.

NOTE: The eLearning folder for LA submission will remain open beyond the due date but will indicate how many days late an assignment was submitted where applicable. The dropbox will be inaccessible seven days after the due date by which time no more credit can be received for the assignment.

The penalty for late submissions as stated in the course syllabus will be applied in grading any assignment submitted late.