**ICS 22 / CSE 22 Fall 2012  
Project #3: *What's Simple Is True***

**Due date and time:** *Wednesday, November 7, 11:59pm*

*This project is to be done individually*

***Introduction***

When I was a kid, one of my teachers introduced me to a computer for the first time — a [Radio Shack TRS-80 Model I](http://en.wikipedia.org/wiki/TRS-80) (anybody remember it?). Immediately, I was interested. I played little math games and messed around with other "state of the art" educational tools from the early 1980's; as you might imagine, the state of the art wasn't much then, but it was fun.

Then one day, my teacher asked me if I wanted to learn how to write my own programs. I thought it sounded like a great idea. So I picked up a book about a language called BASIC — some of you may have played with it before — and typed in a short program that asked a user for a number of hits and a number of at-bats and printed out a batting average. (Believe it or not, my mother still has a printout of it, including the comment at the top: "My first program, by Alex Thornton." Yes, I commented my first program.) I ran the program, tried it out, and I was mesmerized; the computer did exactly what I asked it to, exactly the way I asked it to. And my lifelong obsession with what I would later know to be computer science began.

BASIC was a good teaching tool for its day: versatile and easy-to-learn. For this project, I've designed a considerably limited version of BASIC called Facile, which supports only eleven kinds of statements. You'll be building a Facile *interpreter*. The project will give you additional practice with inheritance and polymorphism, ask you to implement a linked-list-based stack, and provide you with some design experience, as I've left portions of the program for you to design.

***Reminder: Do not partner up***

For this project, your work is expected to be completed individually, so *do not* partner up, and *do not* use the pair programming technique as in the previous projects. Pair programming is a great technique, but to prepare you for your work in future courses where individual work will be expected, I'd like you to have the opportunity this quarter to work on some of your projects individually.

***The Facile language***

I'll discuss the requirements for your interpreter later in the write-up. First, let's talk about the Facile language.

A Facile program is a sequence of *statements*, one per line. Here's an example of a Facile program:

LET A 3

PRINT A

GOSUB 7

PRINT A

PRINT B

GOTO 10

LET A 4

LET B 6

RETURN

PRINT A

END

.

Each line contains exactly one statement (i.e., there may be no blank lines). Facile assigns a *line number* to each of the lines, where the first line of the program is numbered 1, the second line is numbered 2, and so on. The last line of the program is a period (.) on a line by itself. Execution of a Facile program always begins at line number 1. There is no predefined limit on the number of lines in a Facile program.

**Variables**

A Facile program has 26 variables, named by the capital letters A through Z. Each variable is capable of storing an integer value. Variables do not need to be declared like they do in Java; all 26 of them are created automatically at the beginning of the program, and each is given the value 0 initially.

The value of a variable may be changed with a LET statement. A LET statement changes the value of one variable. Some examples are:

* **LET A 3** - changes the value of the variable A to 3
* **LET Z -9** - changes the value of the variable Z to -9

You can print the value of a variable to the console by using a PRINT statement. A PRINT statement prints the value of one variable, followed by a newline.

So, consider the following short Facile program:

LET A 3

LET Z -9

PRINT A

PRINT Z

.

Its output would be:

3

-9

**Execution of a Facile program**

A Facile program is executed one line at a time, beginning at line number 1. Ordinarily, execution proceeds forward, so that line 1 will execute first, followed by line 2, followed by line 3, and so on. Execution continues until either an END statement is reached, or until it reaches the "." line that appears at the end of the program.

Like any programming language, it is possible in Facile to write programs that execute out of sequence, though the mechanisms are a bit more primitive than they are in a language like Java. A GOTO statement causes execution to "jump" immediately to the given number. For example, the statement **GOTO 4** jumps execution to line 4. Here's an example Facile program that uses GOTO:

LET A 1

GOTO 4

LET A 2

PRINT A

.

In this program, line 1 is executed first, setting the variable A's value to 1. Then the GOTO statement will immediately jump execution of the program to line 4, skipping the second LET. Line 4 prints the value of A, which is 1. So, the output of the program is 1.

A GOTO statement may jump either forward or backward, meaning that the following program is a legal Facile program. See if you can figure out what its output would be. (Remember that the value of a variable that hasn't yet been assigned with a LET is 0.)

LET Z 5

GOTO 7

LET C 4

PRINT C

PRINT Z

END

PRINT C

PRINT Z

GOTO 3

.

GOTO statements are not permitted to jump beyond the boundaries of the program, to lines before line 1 or lines after the "." that completes the program. If such a GOTO statement is encountered while a program is executed, the interpreter terminates with an error message.

**Mathematical operations**

Facile provides the typical mathematical operations that can be performed on variables: addition, subtraction, multiplication, and division. Each operation is provided as a statement that changes the value of the given variable. Here are examples of their use:

LET A 4

ADD A 3

PRINT A

LET B 5

SUB B 3

PRINT B

LET C 6

MULT C 7

PRINT C

LET D 7

DIV D 2

PRINT D

.

In the example above, the ADD statement adds 3 to the value of A, storing the result in A. So, printing A will display 7 on the console. The output of the program above is:

7

2

42

3

It is important to note that, since all variables in Facile are integers, the DIV statement implements integer division, meaning that its result is the floor (or integral part) of the quotient. So, in the example above, 7 / 2 = 3. The second operand may not be zero, meaning that the statement **DIV A 0** is illegal. When a Facile program encounters a division by zero, it immediately terminates with an error message.

**The IF statement**

Facile provides an IF statement, which acts like a conditional GOTO. It compares the value of some variable to some value, and jumps execution of the program to the given line number if the comparison is true. The comparison can use one of the typical relational operators: <, <=, >, >=, = (equal to), or <> (not equal to).

LET A 3

LET B 5

IF A < 4 THEN 5

PRINT A

PRINT B

.

In the program above, the variables A and B are given the values 3 and 5, respectively. An IF statement then compares A to 4. Since A is less than 4, execution jumps to line 5. B's value is printed out. So this program's output is simply:

5

The IF statement in Facile is substantially less flexible than its Java equivalent. In an IF statement in Facile, the token IF must be followed by exactly five tokens. The first must be the name of a variable. The second must be one of the relational operators (<, <=, >, >=, =, or <>). The third must be an integer constant. The fourth must be the word THEN. The fifth must be a line number. They behave in the way you might expect. For example: **IF C <> 0 THEN 4**means "jump to line 4 if C is not equal to 0".

Like GOTO statements, IF statements are not permitted to jump beyond the boundaries of the program. An attempt to do so should cause the Facile program to terminate with an error message.

**Subroutines**

There are no methods or functions in Facile, but there is a simplified mechanism called a *subroutine*. A subroutine is a chunk of Facile code that can be "called" by issuing a GOSUB statement. GOSUB is much like GOTO; it causes execution to jump to a particular line. However, GOSUB also causes the Facile program to remember where it jumped from. Subsequently, when a RETURN statement is reached, execution continues at the line following the GOSUB statement that caused the jump. Here's an example:

LET A 1

GOSUB 6

PRINT A

PRINT B

END

LET A 2

LET B 3

RETURN

.

In the program above, line 1 is executed first, setting the value of A to 1. Next, a GOSUB statement is reached. Execution jumps to line 6, but Facile also remembers that when a RETURN statement is reached, execution should jump back to the line following the GOSUB — in this case, line 3. Line 6 is executed next, setting A to 2, then line 7 sets B to 3. Now we reach a RETURN statement, causing execution to jump back to the line number that we're remembering — line 3. Line 3 prints the value of A (which is 2), then line 4 prints the value of B (which is 3). Next, we reach line 5, which is an END statement, so the program ends.

Subroutines can be used very similarly to Java methods, except they do not take parameters or return a value. Consider the following example, which contains a subroutine that prints the values of A, B, and C each time it's called:

LET A 3

GOSUB 10

LET B 4

GOSUB 10

LET C 5

GOSUB 10

LET A 1

GOSUB 10

END

PRINT A

PRINT B

PRINT C

RETURN

.

Subroutines may call other subroutines, meaning that two or more GOSUB's may be reached before a RETURN is reached. The rules for this are very similar to methods that call other methods in Java; for each GOSUB that is reached, Facile will remember the line to which it should return. When a RETURN is reached, execution will move to the line remembered from the *most recent* GOSUB. Here's an example:

LET A 1

GOSUB 7

PRINT A

END

LET A 3

RETURN

PRINT A

LET A 2

GOSUB 5

PRINT A

RETURN

.

In this example, execution begins at line 1 by setting the variable A to 1. Next, we jump to line 7 with a GOSUB, but remember that we should jump back to line 3 when we encounter a RETURN. Line 7 prints A (which is 1), then line 8 changes A's value to 2. Now we've reached line 9, which is another GOSUB statement. At this point, execution will jump to line 5, but we'll also need to remember to jump back to the line following this GOSUB — line 10 — when we reach a RETURN. But we also need to remember the line from the previous GOSUB — line 3.

Line 5 sets A to 3, then we encounter our first RETURN statement. We're remembering two lines — line 3 and line 10. But line 10 is the most recently remembered line, so execution jumps to line 10. Line 10 prints A (which is 3). Now we encounter another RETURN statement on line 11. We're remembering the line 3 from the first GOSUB. So execution jumps to line 3, printing A (which is still 3), then ending the program on line 4.

So, the output of this program is:

1

3

3

Like GOTO statements, GOSUB statements are not permitted to jump beyond the boundaries of the program, to lines before line 1 or lines after the "." that completes the program. If such a GOSUB statement is encountered while a program is executed, the interpreter terminates with an error message.

It is also an error for a RETURN statement to be encountered when there has been no previous GOSUB. The Facile program will immediately terminate and print an error message in this case, as well.

**Whitespace**

While Facile programs may not have blank lines in them, the amount and placement of blank space between the words on each line is considered irrelevant. So, the following is a legal Facile program:

LET Z 5

GOTO 7

LET C 4

PRINT C

PRINT Z

END

PRINT C

PRINT Z

GOTO 3

.

***Do you want to experiment with Facile?***

An *interpreter* is a program that is capable of executing a program written in some programming language. To give you the ability to experiment, I've implemented a Facile interpreter for Windows already. (For those of you who don't ordinarily use Windows, remember that our machines in the ICS labs run Windows, so you'll have ample opportunity to experiment with Facile. You might even want to "pair program" while you experiment.) This [Zip archive](http://www.ics.uci.edu/~thornton/ics22/LabManual/Simple/Facile.zip) contains the interpreter (Facile.exe) and all of the Facile programs that appear in this write-up, along with a few additional ones that demonstrate fatal errors (division by zero, a RETURN statement without a corresponding GOSUB, and a GOTO to a non-existent line). Feel free to write your own, as well. Unzip the archive into one folder, then double-click the program. From there, it's fairly self-explanatory. A word of warning about my interpreter: I wrote it without making a serious attempt at handling syntax problems, so it assumes that the input file is a legal Facile program. If you attempt to run an input file that is not legal Facile, you may see the message "ERROR IN PROGRAM", but it's also possible that my interpreter may simply crash.

I'm providing this interpreter so you can experiment with the language as you have questions about it. Once you're comfortable with it, it'll be your turn to implement a Facile interpreter. (Bear in mind that my Facile interpreter implements much of the optional work described in the "Additional challenges" section below, but it will behave correctly on the samples given in this write-up.)

***The program***

For this project, you'll be building your own *Facile interpreter*, which is a program that is capable of executing a Facile program, generating the correct output according to the specification in the previous sections. Since you're already familiar with Java, you'll write your Facile interpreter in Java. (Since Java runs on many operating systems, that means, once completed, you'll be able to use your interpreter to run Facile programs on Windows, Mac OS X, Linux, Unix, and several other platforms. Not too shabby!)

I will be providing you with a starting point, but I will not be providing a complete skeleton for your program, so you'll have a bit more design work to do this time around. Don't worry; plenty of design advice is available below and, of course, we're happy to answer questions as you encounter problems.

The Facile interpreter that I've provided runs in a (very simple) graphical user interface. Your program, on the other hand, should read one Facile program from an input file, then execute it, writing any output from the Facile program to the console (i.e., **System.out**).

So that we can keep everything straight during the grading process, please write your **main()** method in a class called **Facile**, so that we can run your program using the following command:

java Facile program1.f

where the name of the input file is specified as a command-line argument to the program. If we ran your interpreter with the command above, it would execute the Facile program in the file **program1.f**.

***Starting point***

As with the previous projects, I'm providing some code to get you started. This time, I'm providing some of the underlying code, but I'll let you design part of the program on your own, including the class that contains the **main()**method. Bear in mind that your **main()** method will need to be in a class called Facile, and that **main()** method must expect the filename of the Facile program to be passed as a command-line argument, as discussed in the previous section.

The provided code is available as a [Zip archive](http://www.ics.uci.edu/~thornton/ics22/LabManual/Simple/Simple.zip).

***How does an interpreter work?***

A typical interpreter will execute a program one statement at a time, keeping track of what we might call the *program state* as it goes along. In the case of a Java interpreter, you might imagine that there would be quite a bit of work to be done. The interpreter would need to keep track of all of the objects — creating new ones and garbage-collecting old ones as necessary — as well as maintain the "call stack," along with various other tasks required by Java programs. Implementing an efficient, complete Java interpreter is a project that would easily take many programmer-years.

A Facile interpreter is a much simpler program, since Facile is a much simpler programming language. Your interpreter will need to execute a Facile program one statement at a time, updating the program state as necessary, until either an END statement or the "." is reached. (The "." can simply be treated as an END statement, if you'd like.) The program state consists of the following information:

* what line of code is currently executing (you might call this the *program counter*)
* the integer value in each of the 26 variables
* the line numbers remembered because of any GOSUB statements; since each RETURN jumps back to the line following the most recent GOSUB, it makes sense to store these line numbers in a stack

Each statement has a different effect on the program state. For example, a LET statement will cause the value of one of the variables to change, then cause the program counter to be incremented (since, after a LET statement, execution continues on to the next statement), a GOTO statement will cause the program counter to be changed to the line number specified in the statement, and so on.

***Reading the program from an input file and representing it in memory***

Your program will need to begin by reading the Facile program from an input file and representing it in memory. There are a number of ways to solve this problem. One way is to read the program into memory as a collection of Strings, with each of the Strings containing one line of the input program. Every time a particular line is executed, it would need to be parsed (to see what kind of statement it was), then executed. As you might imagine, this is a terribly inefficient way to implement an interpreter, since the same statement may need to be parsed over and over again. You are not permitted to use this approach for your interpreter.

A better approach — one that I'm requiring you to use instead — is to read the input program once, parse it once, and represent it as objects. Inheritance and polymorphism provide a very natural design approach for this problem.

* An abstract class called Statement contains any functionality common to all statements. I've provided such an abstract class as part of the starting point. The only common functionality for all statements is that they can be executed, though what happens when they are executed is different depending on the type of the statement. This is represented in the Statement class I provided as an abstract method called **execute()**.
* For each kind of statement (e.g., LET, PRINT, etc.), a subclass of Statement can be designed (e.g., LetStatement, PrintStatement). Each class contains fields that store the information needed to execute the statement. In the case of a LET statement, for example, the necessary information is the name of the variable and the value to assign into it. Also needed in each of these Statement subclasses is an actual implementation of the **execute()**method. As an example, I've provided a partially-complete LetStatement class in the starting point. You'll need to design and implement the others.

There needs to be code that can parse the input file and create the appropriate sequence of Statement objects. As part of the starting point, I've provided a skeletal implementation of a class called Parser that does just that: it reads the input file and returns an ArrayList<Statement> containing all the statements in the program. Note that line numbers in Facile start at 1, not 0, so I suggest storing null as the first element in the ArrayList, then storing the actual statement objects with indices beginning at 1. (An alternative, storing the statements beginning at index 0, will require the error-prone practice of adding or subtracting one when converting between line numbers and ArrayList indices, which can easily lead to chaos.)

You may assume that the input file contains a syntactically legal Facile program. It's acceptable for your program to either print an error message, ignore lines that aren't understood, or even crash in the event that it's given an input file that is not legal Facile. (It's a good thing Java compilers don't behave this way.) We will only test your interpreter with syntactically legal Facile programs, though the programs may have run-time errors in them. As was discussed above, there are three kinds of run-time errors: division by zero, a RETURN statement without a corresponding GOSUB, and a GOTO/GOSUB/IF..THEN to a line outside of the boundaries of the program. Your interpreter will need to behave reasonably in these cases, by printing a meaningful error message and terminating gracefully.

***Designing your interpreter***

As the size of a program increases, one of the most difficult obstacles that programmers face is the need to separate their concerns. One of the primary strategies that programmers use to separate their concerns is to break a large program into a set of smaller pieces. The obvious mechanism for breaking up a program in an object-oriented language such as Java is the use of classes.

It is especially difficult for novice programmers to keep their concerns separate. The temptation is always to try to think about the complete picture, since this strategy works well for the short programs that you write when you're first starting out. As programs become larger, confusion naturally sets in, as the complete picture can be difficult to keep in your brain all at once. Even moderately small Java programs are typically built out of many classes and encompass a great deal of complexity. My complete interpreter has approximately 30 classes. (Yours may have fewer, since I also had to write classes that implemented the graphical user interface, and also implemented all of the optional work.) Now, before you freak out, bear in mind that many of them are relatively short. I opted to write more classes with less code in each, so that I could concentrate my efforts on each one largely in the absence of the others. This project will encourage you to begin thinking about your programs the same way, which will give you the ability to write much larger programs than you could before.

The main tasks that your program must perform are:

* Read the contents of the input file, parsing each line, and storing an object into memory that represents the Facile statement appearing on that line.
* Create a representation of the initial program state, then begin executing the program one statement at a time. The execution of each statement will cause the program state to be changed, and may also cause output to appear on the console.
* Continue executing the program until an END statement or the "." is reached.

I suggest breaking up your program in the following way:

* **Facile**. This class will contain your **main()** method. As I did in the main() methods that I've provided in previous projects, I suggest that it does very little, other than creating an Interpreter object, passing the program's command-line argument to the constructor. It should print an error message if no command-line argument is passed to the program.
* **Interpreter**. This class will oversee the execution of the interpreter on one input file. Interpretation requires following the sequence of steps above: parsing the input file, creating an initial program state, then executing one statement at a time until the program ends. Most of the actual work is delegated to other classes, with Interpreter acting as a manager.
* **Parser**. This class parses the input file and returns an ArrayList of objects representing statements.
* **Stack<E>**. A generic stack, which you'll use to store the return points from GOSUB statements. Since there is no maximum number of GOSUB's that may be encountered during a program, you must implement your stack as a linked-list-based stack.
* **ProgramState**. This class represents the state of an executing Facile program. It contains the program counter, the values in each of the 26 variables, and a stack of return points from GOSUB statements.
* **Statement**. This (abstract) class represents a Facile statement. Subclasses such as LetStatement, PrintStatement, etc., implement the actual statements.

I've provided portions of many, but not all, of these classes in the starting point.

It's a good idea to build as many of the underlying pieces as you need to implement a couple of the statements, say LET and PRINT, first. Afterward, add new kinds of statements one or two at a time, making any changes required in the underlying pieces.

***Implementing your stack generically***

As part of the starting point for this project, I've provided a skeleton for a generic Stack<E> class. You are required to implement your stack generically using this skeleton. It is required that you implement it as a linked list. I suggest using your LinkedList<E> class from the previous project as the underlying implementation, though you may rebuild the linked list functionality directly into the Stack<E> class if you prefer.

***How to use a Stack<E> to store integers***

Generic classes are very versatile, in that they can be used to store any kind of object. A Stack<String> stores Strings, while a Stack<ArrayList<Voter>> stores ArrayLists of Voters. However, there is one wrinkle: values of primitive types like **int** and **char** are not objects! This brings up the question of how to store **int**s into generic classes such as ArrayList or Stack, since it isn't legal to create an ArrayList<int> or a Stack<char>.

There are classes, such as Integer and Character, in the Java library that define object wrappers for each primitive type. An Integer object is an object that stores an **int** within it. Java 5.0 is very adept at converting **int**s to Integer objects and back again automatically, using a new feature called *autoboxing* and *autounboxing*. Autoboxing means automatically wrapping a primitive value into an object; autounboxing means automatically unwrapping an object and giving you back the primitive value inside of it.

The tricky part is knowing how to declare a generic class appropriately to allow it to store these "boxed" primitive values for you. Suppose you want to store an ArrayList of integers. Here's how you create one:

// Notice the use of Integer instead of int

ArrayList<Integer> a = new ArrayList<Integer>();

Thanks to automatic conversions from **int** to Integer and back, you can now treat the ArrayList as though it contained **int**s instead of Integers. Here are a couple of examples:

// This statement adds the int 3 to the ArrayList. It turns out that

// Java will automatically wrap the int with an Integer object, but this

// is only relevant from a performance perspective (wrapping takes time).

a.add(3);

// When fetching values out of the ArrayList, you can fetch an int instead

// of an Integer object, with Java automatically unwrapping the object for

// you.

int i = a.get(0);

This approach will be handy in your program, since you'll need a stack of integers. I suggest declaring it as a Stack<Integer>, after which you can essentially treat it as though it were really a Stack<int>.

***Command-line arguments: passing parameters to an entire program***

**What are command-line arguments?**

In Java, methods can take *parameters*, which allow the caller to configure what job the method should do. For example, the get() method in the ArrayList class allows us to ask for one of the elements in the ArrayList; it requires an integer parameter, which specifies the index of the element that the caller wants. If we want the first element in the list, we call get() with a parameter of 0.

Programs, too, can take parameters. Why we want them is the same reason why we want method parameters: if programs can take parameters, we can use them to configure how the program behaves. In previous work, you may have used a program called **javac**, a Java compiler, to compile your Java source files. Say, for example, you had a file called MusicList.java and you wanted to compile it; from the command line, you'd run this command:

javac MusicList.java

In this case, **javac** is the name of the program. If you just typed **javac**, you'd be telling the Java compiler to run, but you wouldn't be telling it what file you wanted to compile! After the name of the program, the remaining elements of the command are called *command-line arguments*; in this case, **MusicList.java** is a command-line argument that's interpreted by the program as the name of the file to be compiled.

Java programs can take command-line arguments, as well. When you run a Java program from the command line using the **java** command, you typically write a command line this:

java Facile

where, in this case, **Facile** is the name of the class that contains your main() method.

If you want to pass a command-line argument to a Java program from the command line, you do so by just adding them to the command, just like you do when you use **javac**. For example, if you want to pass the filename **prog1.f** as a parameter to your Facile interpreter, you could do so from the command line like this:

java Facile prog1.f

The command line arguments are made available to your program, which can use them to configure its behavior.

**Accessing command-line arguments within a Java program**

Have you ever wondered why the signature of the main() method is this?

public static void main(String[] args)

In particular, have you ever wondered what the **args** array is all about? Well, it's no mystery anymore: **args** contains the command-line arguments.

For example, if you run a Java program this way on the command line:

java MyProgram alex is happy today

the MyProgram class' main() method will be called, and its **args** array will look like this:

|  |  |  |  |
| --- | --- | --- | --- |
| ***0*** | ***1*** | ***2*** | ***3*** |
| alex | is | happy | today |

So, if you accessed **args[0]** in main() in this example, you'd get **alex**; if you access **args[1]**, you'd get **is**, and so on. Also, since arrays can tell you their length if you access their **length** field, **args.length** will tell you how many command-line arguments there were. In the example above, **args.length** will be 4.

In your program, you'll want to pass the value of args[0] — which is expected to be the filename — from your main() method into the Interpreter class, which can then use that information to know what file it should be parsing and interpreting. You'll also want, in your main() method, to check whether the number of arguments in anything other than 1; if it is, you'd want to print an error message and return from main() without doing anything else.

**Setting the command-line arguments in Eclipse**

There's one other issue for us, since we use Eclipse, rather than the command line. When you run a Java program from within Eclipse, you don't type a command on the command line; you right-click a file in the Package Explorer and select **Run As**, then **Java Application**. So, how do you set the command-line arguments? The answer is that you have to run your program a little bit differently.

To run your program with command-line arguments, use this procedure instead:

* As you would do normally, right-click on the .java file in your Package Explorer window that contains your main() method. (In this case, Facile.java.)
* On the ensuing menu, select **Run As**, also as you would do normally.
* Instead of then selecting **Java Application**, select **Run Configurations...**.
* This will cause a window to pop up with a number of options that you can set. (Don't let yourself get lost in all the options; there's only one thing you'll want to set.) On that window, choose the **Arguments** tab.
* In the **Program arguments** field, type your arguments. In this case, type the name of the filename you'd like your interpreter to run (e.g., **prog1.f**).
* Click the **Run** button in the lower-right corner of the window. Eclipse will now execute the program as though you had typed this command from the command line:
* java Facile prog1.f

To allow Eclipse to find your Facile files, place them in the folder within your Eclipse workspace that corresponds to this project. So, for example, if your project is called **Project3**, find the folder called **Project3** within your Eclipse workspace and put the Facile files there. (Note that you shouldn't put them in the **bin** or **src** folders within your project.)

Be aware that, once you've done this, every time you run your program, it will have the same arguments until you use this procedure to change them. Other programs, like the ones you wrote in previous projects, will not be affected, though.

***Testing***

To satisfy the testing portion of this project, you're required to use JUnit to implement unit tests, as you did in the previous project. Your unit tests should, at minimum, test the Stack<E> and ProgramState classes, though you may include tests for other classes if you wish. (You may well find that additional testing helps you to get your program working.)

Remember that unit tests are separate from the program whose classes they test, so executing the complete Facile interpreter should not execute the tests (nor vice versa).

***Facile quick reference***

Here is a list of all of the Facile statements that should be supported by your interpreter, with a brief description of the effect of each. In each of the statements below, *var* may be the name of a variable (A, B, C, ..., Z), *int* may be an integer constant (e.g., 1, -3, 15), and *linenum* may be a line number (1..1000).

|  |  |
| --- | --- |
| ***Statement*** | ***Description*** |
| LET *var* *int* | Changes the value of the variable *var* to the integer *int*. |
| PRINT *var* | Prints the value of the variable *var* to the console. |
| ADD *var* *int* | Adds *int* to the value of the variable *var*. |
| SUB *var* *int* | Subtracts *int* from the value of the variable *var*. |
| MULT *var* *int* | Multiplies the value of the variable *var* by the integer *int*. |
| DIV *var* *int* | Divides the value of the variable *var* by the integer *int*. |
| GOTO *linenum* | Jumps execution of the program to the line numbered *linenum*. |
| IF *var* *op* *int* THEN *linenum* | Compares the value of the variable *var* to the integer *int* using the relational operator *op* (=, <>, <, <=, >, >=). If the comparison is true, jumps execution of the program to the line numbered *linenum*. If not, this statement has no effect. |
| GOSUB *linenum* | Temporarily jumps to the line numbered *linenum*. A RETURN statement will cause execution to jump back to the line following the GOSUB. |
| RETURN | Jumps execution of the program back to the line following the most recently-executed GOSUB statement. |
| END | Ends the program immediately. |
| . | Special marker that indicates the end of the program text. Behaves as an END statement when encountered. |

***Deliverables***

You must submit all of your **.java** files, including those that were provided to you. Please do not turn in the **.class** files, or other files generated by your development environment.

Follow [this link](http://www.ics.uci.edu/~thornton/ics22/LabManual/SubmittingProjects.html) for a discussion of how to submit your project.

***Limitations***

You must implement your own stack class. Furthermore, you must use a linked-list-based implementation. You may not store the stack elements in an array or ArrayList, and you may not use the existing java.util.Stack class. You also may not use the existing java.util.LinkedList class.

That said, you *may* (and, in fact, should) use an ArrayList to store the objects that represent the statements in the program. Remember that the program is not to be stored in a stack; the stack is for storing return points from GOSUB statements. Of course, you should use a generic ArrayList appropriately specialized (e.g., ArrayList<Statement>).

***Additional challenges***

My Facile interpreter implements some additional features. If you want to work further on your interpreter, you might try supporting these additional features, though they are not required, and no extra credit is offered for them.

Firstly, I included two additional statements:

|  |  |
| --- | --- |
| ***Statement*** | ***Description*** |
| INC *var* | Adds 1 to the value of the variable *var*. For example, the statement **INC A** adds one to the value of A. |
| DEC *var* | Subtracts 1 from the value of the variable *var*. For example, the statement **DEC A** subtracts one from the value of A. |

Including these statements in Facile does not dramatically increase its power, but it does allow for convenient incrementing and decrementing, which can be handy for constructing simple "loops."

I've made one additional improvement in my interpreter that increases the expressiveness of the language quite a bit. Consider a statement such as LET. As defined above, the LET statement sets the value of some variable to some integer constant. But imagine that you wanted to set the value of some variable to be equal to the value of some other variable. Facile, as defined above, does not allow this fundamental operation. But there's no reason it couldn't; and, in fact, my interpreter does.

In many places where an integer constant may normally appear in a Facile program, my interpreter also allows the name of a variable to appear. In the case of PRINT, I also allow an integer constant instead of a variable name. So, for example, these statements may be given to my interpreter:

* **LET A B** - Sets the value of A to be equal to the value of B.
* **PRINT 3** - Prints the integer constant 3 to the console.
* **ADD B C** - Adds the value of C to the value of B, storing the result in B.
* **SUB B C**, **MULT B C**, **DIV B C** - similar to the ADD statement above
* **IF A <= B THEN 4** - Jumps to line 4 if A is less than or equal to B.
* **IF 3 <= B THEN 4** - Jumps to line 4 if 3 is less than or equal to B.
* **IF 4 <= 9 THEN 4** - Jumps to line 4 is 4 is less than or equal to 9.

This is the extent of Facile, as it is supported by my interpreter. If you're interested in discussing approaches for implementing the suggestions above, I'd be happy to chat with you about it.

You might also consider designing and implementing some new statements to accomplish some of these important goals, or others of your own choosing:

* Allow Facile programmers to put comments into their code. (This could bring up an interesting question about the design of the language: how should line numbers be counted if not all lines contain code?)
* Define additional variables that can store string values instead of integers. The BASIC language names such variables with trailing $ characters. So you might have the variables A$, B$, C$, ..., Z$, each of which is capable of storing a string.
* Add a statement, or perhaps a variant of the PRINT statement, to output a string of text (a string literal or the value of a string variable) to the console.
* Allow the IF statement to compare two string variables, or to compare a string variable to a string literal.
* Add a statement to read an integer and/or string from the console and store it in a variable.
* Testing section rewritten to require JUnit by Alex Thornton, Winter 2012.
* Testing section added back into the assignment by Alex Thornton, Winter 2007.
* Clarification to the GOTO/GOSUB error condition, specifying that the THEN part of IF statements is to be included, made by Alex Thornton, along with necessary changes to provided interpreter, Spring 2006.
* A new error condition (GOTO/GOSUB to non-existent line) added by Alex Thornton, along with necessary changes to provided interpreter, Winter 2006.
* A couple of very minor changes made by Alex Thornton, Fall 2005.
* Some minor changes introduced by Alex Thornton, Spring 2005.
* Assignment restructured (to require the use of an object-oriented solution, while no longer requiring the testing portion) by Alex Thornton, Fall 2004.
* Almost all of the work that was previously optional is now required, along with other heavy modifications, as well as improvements to the given interpreter by Alex Thornton, Winter 2003.
* Originally written by Alex Thornton, Fall 2002.