This assignment is to be completed in pairs

Assignment Two
Virtual Machine

**Software Design and Programming and
Software and Programming III**

Spring Term

2017

Please read this *entire* document carefully. If, after a careful reading, something seems ambiguous or unclear to you, then communicate this to the teaching team immediately. You should start this assignment as soon as possible. Do not wait until the night before the assignment is due to tell us you dont understand something, as our ability to help you will be minimal/non-existent.

The sample code mentioned in the text can be found on the module repository under the `cw-two` directory. This coursework assignment must be completed using the Scala programming language.

Learning Goals

- To understand the problem domain.
- To apply appropriate constructs in Scala to solve the problem.
- To understand and implement the factory pattern.
- To understand and implement the singleton pattern.
- To understand and implement the adapter pattern.
- To understand and implement the command pattern.
- To understand and apply appropriate git commands.
- To work in a team consisting of two people.
- To document your code clearly.
- To test your code thoroughly and develop your code using test-driven development techniques.

Overview

This assignment examines your understanding of Scala and *Design Patterns* [1]. This assignment extends several of the concepts that you encountered in the first coursework assignment.

You will complete the implementation of a stack-based virtual machine that executes a small set of "bytecode" instructions. Just like the Java Virtual Machine (JVM) that executes JVM bytecode (the representation your Scala/Java programs are compiled to), the virtual machine that you implement will execute a small set of bytecode instructions represented as textual commands that manipulate the state of a virtual machine. These commands/instructions perform operations on a stack-based virtual machine to execute simple computations. Here is an example of a simple program (in textual format) that adds two numbers, increments the result, then prints the result to the console:

```
iconst 4
iconst 5
iadd
print
```

- The `iconst` instruction takes a single integer argument and pushes it onto the stack of the virtual machine.
- The `iadd` instruction pops the top two integer values from the stack, adds them, and pushes the result.
- The `print` instruction pops the value on the top of the stack and prints it to the console.

Thus, executing this simple bytecode program will produce `9` as its output.

As part of the implementation of the virtual machine you will need to use the *command*, *adapter*, *factory*, and *singleton* design patterns. You will use the *command* pattern to implement the bytecode operations, the *adapter* pattern to adapt a third party vendors textual bytecode parser (which you will also implement) to a virtual machine parser that parses the bytecode integer representation into bytecode objects, and the *factory* and *singleton* patterns to create bytecode objects for the testing harness.

Instructions

This virtual machine is simple in that it only has a working stack to perform computations. Each instruction described below has a textual representation, a byte value, and the semantics of the instruction (what it does when it is executed). The byte value is used by the virtual machine to identify which instruction it is executing. Each instruction assumes the existence of a virtual machine `VM` that is used to define the corresponding semantics.

iconst NUM : The `iconst` instruction pushes the integer value `NUM` on the virtual machine stack.

`VM.push(NUM)`

[1] `http://www.oodesign.com`

iadd : The `iadd` instruction pops the top two values from the virtual machine stack and pushes the result.

```
VM.push(VM.pop() + VM.pop())
```

isub : The `isub` instruction pops the top two values from the virtual machine stack and pushes the result.

```
VM.push(VM.pop() - VM.pop())
```

imul : The `imul` instruction pops the top two values from the virtual machine stack and pushes the result.

```
VM.push(VM.pop() * VM.pop())
```

idiv : The `idiv` instruction pops the top two values from the virtual machine stack and pushes the result.

```
VM.push(VM.pop() / VM.pop())
```

irem : The `irem` instruction pops the top two values from the virtual machine stack and pushes the result.

```
VM.push(VM.pop() % VM.pop())
```

ineg : The `ineg` instruction pops the the top value from the virtual machine stack, negates it, and pushes the result.

```
VM.push(-VM.pop())
```

iinc : The `iinc` instruction pops the the top value from the virtual machine stack, increments it, and pushes the result.

```
VM.push(VM.pop()+1)
```

idec : The `idec` instruction pops the the top value from the virtual machine stack, decrements it, and pushes the result.

```
VM.push(VM.pop()-1)
```

iswap : The `iswap` instruction pops the top two values from the virtual machine stack and pushes them in the opposite order, effectively swapping the top two values of the stack.

```
x = VM.pop();
y = VM.pop();
VM.push(x);
VM.push(y)
```

idup : The `idup` instruction pops the top value from the stack and pushes it twice onto the stack (duplicates the top value).

```
x = VM.pop();
VM.push(x);
VM.push(x).
```

print : The `print` instruction pops the top value from the stack and prints the value to the console.

As you can see, the operations themselves are rather simple. We do not list the actual bytecode *byte* representation as they are derived from a list of the instructions in the code. Another important observation is that the semantics of each instruction has access to the virtual machine itself. This is made explicit in the description of the implementation below.

Note, that you should assume that the order of evaluation of the above operators (`+`, `-`, `*`, `/`, `%`) first evaluates its left operand followed by its right operand. That is,

```
VM.push(VM.pop() + VM.pop())
```

is the same as

```
x = VM.pop();
y = VM.pop();
VM.push(x + y)
```

This detail is important in how you implement the semantics of the bytecode.

Test Files

In the `src/test/scala` directory, we provide `ScalaTest` test suites that will help you while developing your code. We recommend you run the tests often and use them to help create a checklist of things to do next. You should be aware that we do not provide you with the full test suite we use when grading.

We recommend that you think about possible test cases and add new test cases to these files as part of your development process. Simple tests to add will consider questions such as:

- Do your methods handle edge cases such as integer arguments that may be positive, negative, or zero? Many methods only accept arguments that are in a particular range.
- Does your code handle unusual cases, such as empty or maximally-sized data structures?
- ...

To build good test cases, think about ways to exercise functions and methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. Note that we will not be looking at your test cases (unless otherwise specified by the assignment documentation); they are just for your use and will be removed by the *auto-grader* during the evaluation process.

If you modify the test cases we have provided or you add your own tests, it is important to know that they will not be used. The auto-grader will use its own copy of the public and private tests. If you modify any source files in the `src/test/scala` directory your changes will not be reflected in the grading of your submission.

Before submitting your work, make sure that your program compiles with, and passes all of, the original tests. If you have errors in these files, it means the structure of the files found in the `src` directory have been altered in a way that will cause your submission to lose some (or all) of its marks.

Project Structure

Your project should (usually) contain the following root items:

src/main/scala : This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

src/instructor/scala : This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder. The auto-grader will replace your submitted `src/instructor` directory with the original during the evaluation process. If you make changes or add/remove anything it can lead to problems in your submission and will result in a failing grade for the assignment.

src/test/scala : The test folder where all of the public unit tests are available.

build.sbt : This is the build definition file. It provides build information to activator to build your project. Do not remove.

.gitignore : This is a special file used by git source control to ignore certain files. You should not touch this file and please do not remove it; this file may not be present.

Testing

As mentioned previously, you are provided with a set of unit tests that will test various aspects of your implementation. You should get into the habit of running the tests frequently to see how you are doing and to understand where you might be going wrong. The `ScalaTest` testing framework command is built-in to the `sbt` tool and you can easily run the tests by issuing the following command from the command line (Mac/Linux):

```
sbt test
```

For Windows users you would issue the following command from the command window:

```
sbt.bat test
```

This will compile your code and run the public `ScalaTest` unit tests. After you compile and run the tests you will notice that a `target` directory has been created. The `target` directory contains the generated class files from your source code as well as information and results from the tests.

Another very useful approach to test and play with the code you write is the sbt/Scala console. You can run the console with this command (Mac/Linux):

```
sbt console
```

For Windows users you would issue the following command from the command window:

```
sbt.bat console
```

This will load up the Scala REPL (read-eval-print-loop). You can type code directly into the console and have it executed. If you want to cut and paste a larger segment of code (e.g., function declaration) you simply type `:paste` in the console, then paste in your code, then type `control-D`.

Editors and IDEs

You are welcome to use any editor or IDE of your choice.

Part I: Getting started

After you clone the provided files you should take some time reviewing the code and comments. There are three main components that your will need to extend/implement to provide a working virtual machine implementation: `vendor`, `bc`, and `vm`.

The Vendor Component (package `vendor`)

The *vendor* component defines an interface to the "vendor" parser that can parse a bytecode program as a textual representation into a sequence of instructions representing those instructions. For this assignment you are to assume that this is code coming from a third party library that you will need to adapt to your virtual machine implementation. (Of course, you will also play the part of the vendor and provide an implementation as well.)

The vendor component resides in the `vendor` package inside the `src/instructor/scala` directory. This package contains the two files outlined below:

Instruction.scala : This file provides two definitions.

The first definition is the class `InvalidInstructionFormatException` — this exception is used to indicate an invalid instruction format when a textual representation of a bytecode program is parsed (translated from text to `Instruction` objects).

The second definition is the class `Instruction`. It represents an instruction found in a textual representation of a bytecode program. Every instruction has a name (e.g., `iconst`, `iadd`, etc.) and a vector of arguments. Fortunately, in our small set of instructions only the `iconst` instruction has an argument; therefore the other instructions the instruction arguments will be empty.

ProgramParser.scala : This file provides a single trait, `ProgramParser` that you will need to extend/implement. It defines a type alias `InstructionList` that is a vector of `Instruction` objects. An `InstructionList` represents a list of instructions found in a textual representation of a bytecode program. It also defines two abstract methods, `parse` and `parseString`, for parsing/reading a program contained in a file or string respectively. These two methods both produce a list of instructions.

The Bytecode Component (package `bc`)

The `bc` component defines an interface that the virtual machine will use to represent bytecode instructions. Of course, this differs from the vendors instruction representation which necessitates the use of an adapter. The bytecode component resides in the `bc` package inside the `src/instructor/scala` directory. This package contains the three files outlined below.

ByteCode.scala : This file provides three definitions.

The first definition is the `InvalidBytecodeException` class. This is used to indicate a problem with a bytecode.

The second definition is the `ByteCodeValues` trait. This contains useful definitions that are extended by several other classes/traits/objects to import the contained definitions into their scope. In particular, it provides a vector of all the names of the bytecode instructions that the virtual machine will support. It also includes a mapping from bytecode names to their byte values. The byte values simply correspond to the position of the bytecode names in the vector of names.

The third definition is the `ByteCode` trait. It extends `ByteCodeValues` to give it access to the definitions mentioned above. It defines the abstract code `value` which is the actual byte value of this bytecode and the abstract `execute` method that will execute the bytecode on the given `VirtualMachine` and return a new `VirtualMachine`. Note, that this is a direct implementation of the command design pattern! We are encapsulating tasks that are to be *executed* by its `execute` method at some point in the future. You will need to implement this trait for each of the twelve bytecode instructions.

ByteCodeFactory.scala : This file contains a single trait, `ByteCodeFactory`, that defines an abstract method `make`. The `make` method will take a byte and an optional integer argument and return a `ByteCode` object. As the name indicates, this uses the factory design pattern to create new objects of class type `ByteCode`. It separates the creation of an object from its actual implementation. You will need to provide an implementation of each of the twelve bytecodes and return them given the corresponding byte and optional argument. This method should throw a `InvalidBytecodeException` if it cant do so. You will need to implement this trait.

ByteCodeParser.scala : This file contains a single trait called `ByteCodeParser` which extends `ByteCodeValues`. It also defines the `parse` abstract method that transforms a vector of bytes into a vector of `ByteCode` objects. It is expected that this trait will use the `ByteCodeFactory` implementation to construct `ByteCode` objects. You will need to implement this trait. For example, given the following vector:

```
Vector(bytecode("iconst"), 4.toByte, bytecode("iconst"), 5.toByte,
       bytecode("iadd"), bytecode("print"))
```

The `parse` method will produce a vector of bytecode objects:

```
Vector(ByteCode, ByteCode, ByteCode, ByteCode)
```

Note, that it only produces four bytecode objects as the `iconst` instructions have arguments.

The Virtual Machine Component (package `vm`)

The `vm` component defines the virtual machine abstraction. It is used to execute a vector of `ByteCode` objects. The virtual machine component resides in the `vm` package under the `src/instructor/scala` directory. This package contains the two files outlined below.

VirtualMachine.scala : This file provides the virtual machine abstraction. In particular, it defines the `MachineUnderflowException` class that can be used to throw exceptions if the stack in your virtual machine does not have enough values to complete an operation. It also defines the `VirtualMachine` trait with useful methods that can be performed by a virtual machine. The `push` and `pop` methods *push* and *pop* values on the virtual machines stack.

Note, that this is an immutable operation, that is, it returns a new virtual machine with a new state rather than changing the state of the stack internally. This is useful as we can easily inspect each virtual machine state in isolation from all the other virtual machine states.

The `executeOne` method executes the first bytecode in the given vector of bytecode objects. It returns a pair containing the new vector of bytecodes, minus the one we executed, and a new virtual machine. This method allows us to step one instruction at a time and inspect the virtual machine state during testing. The `execute` method executes all of the bytecodes, from start to finish, and returns the final virtual machine.

Lastly, the `state` method returns a vector of the state of the stack. This will also help us ensure during testing that the virtual machine executed the bytecode as expected. A virtual machine has executed a given vector of bytecodes properly if

```
vm.state.isEmpty
```

is true. That is, the stack must be empty for it to be considered a valid execution.

VirtualMachineParser.scala : This file contains the definition of the `VirtualMachineParser` trait. This is effectively implementing the adapter pattern. You will use this in combination with the `ByteCodeParser` and the vendors `ProgramParser` to implement the `parse` and `parseString` methods. We are implementing an adapter that connects two other parsers into one that we can use for the virtual machine (which expects a vector of bytecode objects).

Virtual Machine Factory

In addition to the code provided in `src/instructor`, which you must not modify, we provide you the file `factory/VirtualMachineFactory.scala` inside the `src/main/scala/` source directory. You must modify methods in this object to return instances of the classes/objects that you implement. You are not allowed to move or rename this file.

Part II: Implement Vendor Parser

Your first task is to implement `vendor.ProgramParser`. You should take a look at some of the example programs under the `programs` directory. Your implementation should work for these files.

You should also be able to parse bytecode programs contained in strings (i.e., `"iconst 4\nprint"`). After you implement the program parser you need to modify the `factory.VirtualMachineFactory` in the `src/main/scala` directory to return an instance of your implementation from the `vendorParser` method.

Note: your implementation must reside within the `src/main/scala` source directory.

Part III; Implement Bytecodes

Next, you need to implement a bytecode for each of the defined bytecodes in the `bc.ByteCodeValues` trait. You may implement these as classes that extend the `ByteCode` trait.

After you finish implementing each of the bytecodes you should implement the `bc.ByteCodeFactory` that creates new bytecodes given a byte and arguments. If you find it helpful you can override the `toString` method to print out the bytecode names to the console so you can see if your code is working correctly.

After you finish implementing the factory you need to implement the `bc.ByteCodeParser`. You can test this easily from the scala console by providing a vector of bytes and inspecting the returned vector to make sure it is working correctly.

Lastly, modify `factory.VirtualMachineFactory` to return an instance of your bytecode factory and bytecode parser from the `byteCodeFactory` and `byteCodeParser` methods respectively.

Please note: your implementation **must** reside within the `src/main/scala` source directory.

Part IV: Implement Virtual Machine

Next, you need to implement the virtual machine component. To do this you will first need to write an adapter that adapts the vendor parser and bytecode parser with a virtual parser that transforms a textual bytecode program into a vector of bytecode objects. You do this by implementing a `vm.VirtualMachineParser`. That is, you should implement `vm.VirtualMachineParser` as the adapter — use composition of the other two parsers as part of your implementation.

After you complete the virtual machine parser you can implement a `vm.VirtualMachine`. Your virtual machine implementation should use some internal representation for a stack and provide the correct operations defined by the `vm.VirtualMachine` trait.

Lastly, modify `factory.VirtualMachineFactory` to return an instance of your virtual machine from the `virtualMachine` method.

Please note: your implementation must reside within the `src/main/scala` source directory.

Part V: Documentation and Comments

You must provide scaladoc documentation for each of the methods you implement. You should use the scaladoc we provide in `src/instructor/scala` as examples. In addition, you should provide comments to explain code that may require clarity. That is, you should not comment the obvious, but you may want to provides comments to parts of your implementation that may not be entirely clear. Note: scaladoc and comments are different.

Grading

Your submission will be scored on the number of tests that are successfully passed. In addition, the teaching team will be reviewing each submission for proper scaladoc documentation, comments, coding style, and appropriate use of language features. All categories will contribute to your final score for this assignment.

Submission

You should submit this assignment as part of your portfolio. Each member of your team should have a copy of the working code in their repository. The `README.md` file should contain the names of the members of the team. Your repository will be cloned at the stated due date and time.