**Student Name:** Arnold Jiadong Yu

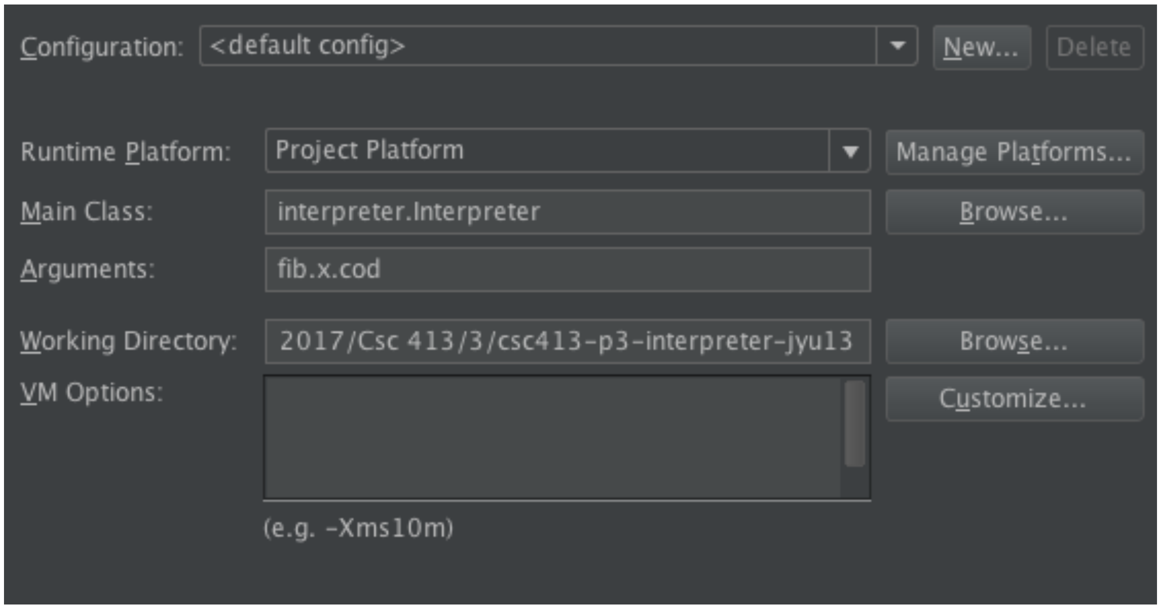
**Student ID:** 917269189

**Class:** Csc413-02

**Link** – https://github.com/CSC-413-SFSU-02/csc413-p3-interpreter-jyu13.git

**Project Introduction**: This is an interpreter project. It demonstrates how java class file is generated and run at java virtual machine. It takes a file as argument. Read each line of the file and generate bytecode instance and store them in an array. It is very similar as MIPS program. All the bytecode classes were implemented in the interpreter.ByteCode file. Main body of RunTimeStack.java and VirtualMachine.java was written. Filled loadCode method in ByteCodeLoader.java. Also filled resolveAddrs in Program.java.

**Compile and execute directions:** This project was built in NetBeans 8.2. Create a new java project with "java project with existing source". Give the project a name then find the directory of the existing source and add the source directory to the project. When compile and execute the project, it needs an argument. Right-click on the program name and choose Set Configuration->Customize. Put the file name as the Arguments and the directory of the file in the working directory.



Configuration: <default config>

Runtime Platform: Project Platform

Main Class: interpreter.Interpreter

Arguments: fib.x.cod

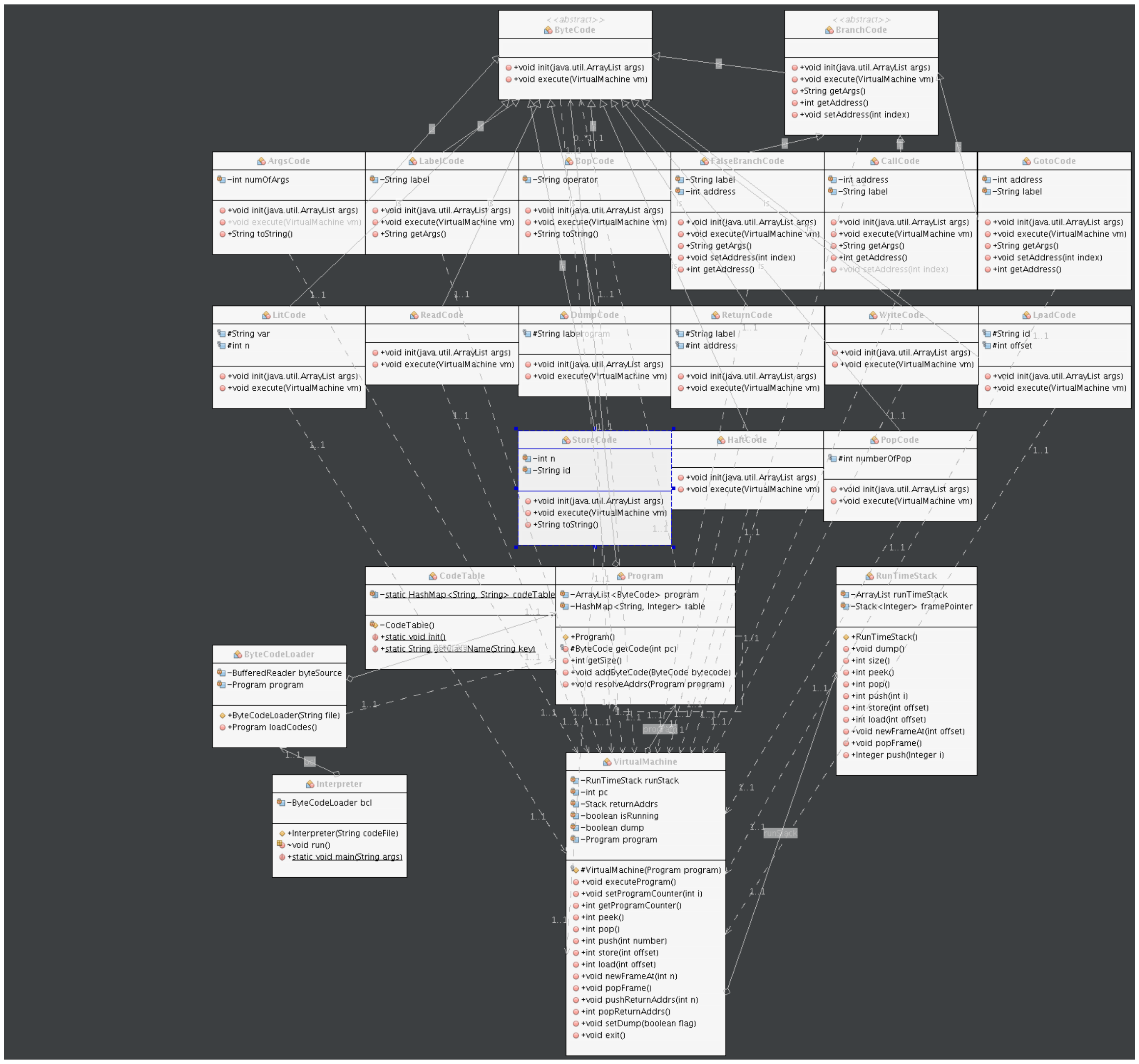
Working Directory: 2017/Csc 413/3/csc413-p3-interpreter-jyu13

VM Options:

(e.g. -Xms10m)

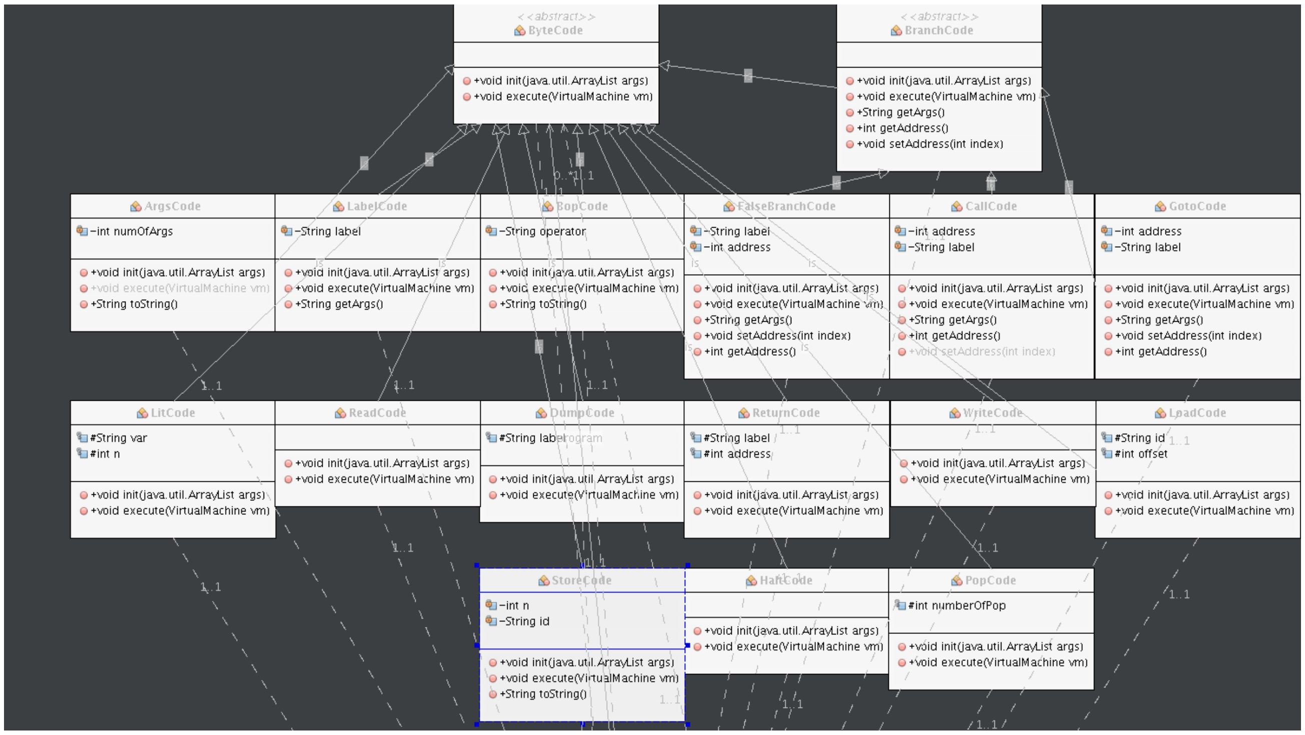
**Assumptions**: I created all bytecode classes first, but didn't implement the code. Then followed the steps were given on the Assignment3.pdf. Worked on ByteCodeLoader.java, Program.java, RunTimeStack.java, and VirtualMachine.java.

**Implementation discussion:** Below is the class diagram.



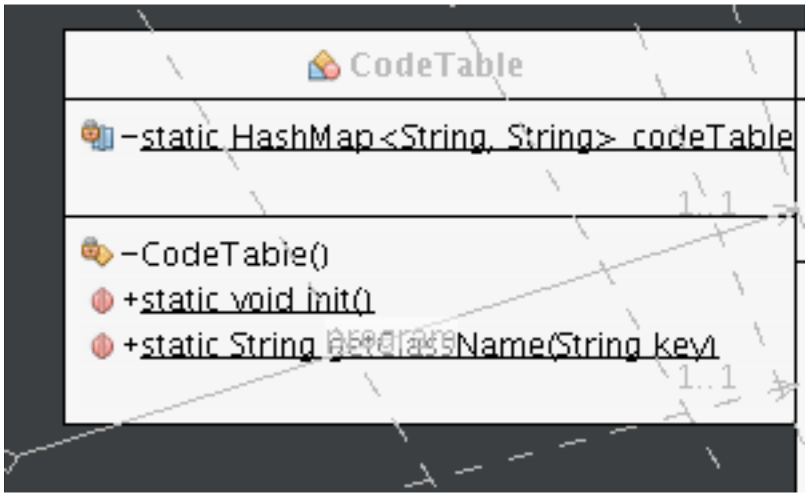
<<abstract>>
ByteCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
<<abstract>>
BranchCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
+int getAddress()
+void setAddress(int index)
ArgsCode
-int numOfArgs
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String toString()
LabelCode
-String label
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
BopCode
-String operator
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String toString()
FalseBranchCode
-String label
-int address
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
+void setAddress(int index)
+int getAddress()
CallCode
-int address
-String label
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
+int getAddress()
+void setAddress(int index)
GotoCode
-int address
-String label
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
+void setAddress(int index)
+int getAddress()
LitCode
#String var
#int n
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
ReadCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
DumpCode
#String label
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
ReturnCode
#String label
#int address
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
WriteCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
LoadCode
#String id
#int offset
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
StoreCode
-int n
-String id
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String toString()
HaltCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
PopCode
#int numberOfPop
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
CodeTable
-static HashMap<String, String> codeTable
~CodeTable()
+static void init()
+static String getClassName(String key)
Program
-ArrayList<ByteCode> program
-HashMap<String, Integer> table
+Program()
#ByteCode getCode(int pc)
+int getSize()
+void addByteCode(ByteCode bytecode)
+void resolveAddrs(Program program)
RunTimeStack
-ArrayList runTimeStack
-Stack<Integer> framePointer
+RunTimeStack()
+void dump()
+int size()
+int peek()
+int pop()
+int push(int i)
+int store(int offset)
+int load(int offset)
+void newFrameAt(int offset)
+void popFrame()
+Integer push(Integer i)
ByteCodeLoader
-BufferedReader byteSource
-Program program
+ByteCodeLoader(String file)
+Program loadCodes()
Interpreter
-ByteCodeLoader bcl
+Interpreter(String codeFile)
~void run()
+static void main(String args)
VirtualMachine
-RunTimeStack runStack
-int pc
-Stack returnAddrs
-boolean isRunning
-boolean dump
-Program program
#VirtualMachine(Program program)
+void executeProgram()
+void setProgramCounter(int i)
+int getProgramCounter()
+int peek()
+int pop()
+int push(int number)
+int store(int offset)
+int load(int offset)
+void newFrameAt(int n)
+void popFrame()
+void pushReturnAddrs(int n)
+int popReturnAddrs()
+void setDump(boolean flag)
+void exit()
1..1

There are two abstract classes in ByteCode, which are ByteCode and BranchCode. ByteCode has two abstract methods that called init() and execute(). All other bytecode classes extend ByteCode. BranchCode was also an abstract class, which overrides two methods from ByteCode and extends with 3 addtion abstract methods. Goto, FalseBranch, Call extends BranchCode since all three classes need to jump to different address. Each init() and execute() method was implemented according to the Assignment3.pdf.



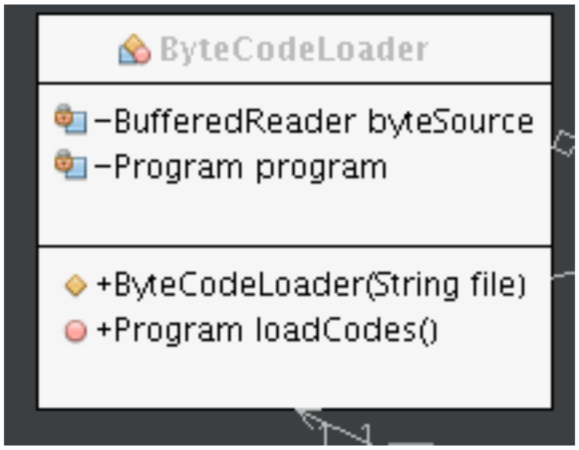
<<abstract>>
ByteCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
<<abstract>>
BranchCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
+int getAddress()
+void setAddress(int index)
ArgsCode
−int numOfArgs
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String toString()
LabelCode
−String label
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
BopCode
−String operator
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String toString()
FalseBranchCode
−String label
−int address
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
+void setAddress(int index)
+int getAddress()
CallCode
−int address
−String label
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
+int getAddress()
+void setAddress(int index)
GotoCode
−int address
−String label
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String getArgs()
+void setAddress(int index)
+int getAddress()
LitCode
#String var
#int n
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
ReadCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
DumpCode
#String label
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
ReturnCode
#String label
#int address
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
WriteCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
LoadCode
#String id
#int offset
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
StoreCode
−int n
−String id
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
+String toString()
HaltCode
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
PopCode
#int numberOfPop
+void init(java.util.ArrayList args)
+void execute(VirtualMachine vm)
1..1

CodeTable.java create a hashmap that generate keys for each bytecode classes. The keys can be used in ByteCodeLoader.java to create new Instances of each bytecode classes.

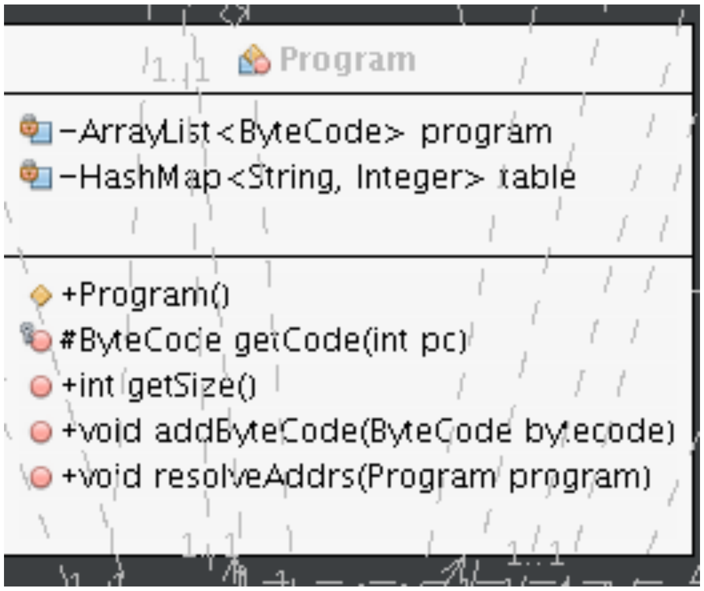


CodeTable
−static HashMap<String, String> codeTable
−CodeTable()
+static void init()
+static String getClassName(String key)
1..1
1..1

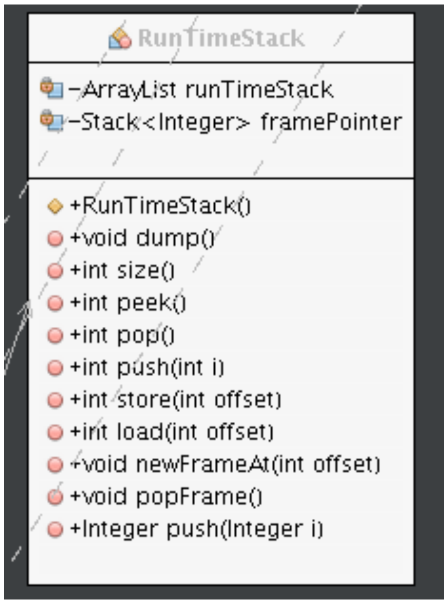
ByteCodeLoad.java create a program object with an arraylist. It stores all bytecode class new Instance in the array. The loadCode() method read each line from the file and initialize each bytecode then put them in the array. It also uses dynamic binding to resolve the address for Goto, Call, and FlaseBranch bytecode.



Program.java has two important method addByteCode(ByteCode bytecode) and resloveAddrs(Program program). First method adds bytecode to the arraylist. Second method changes the address for Goto, Call and FalseBranch bytecode. For example, line 0 : Call continuous. After resolving address it changes to 0. Call 5. 5 represent line 5 that has label continuous and that is the address to jump to.


Program
−ArrayList<ByteCode> program
−HashMap<String, Integer> table
+Program()
#ByteCode getCode(int pc)
+int getSize()
+void addByteCode(ByteCode bytecode)
+void resolveAddrs(Program program)

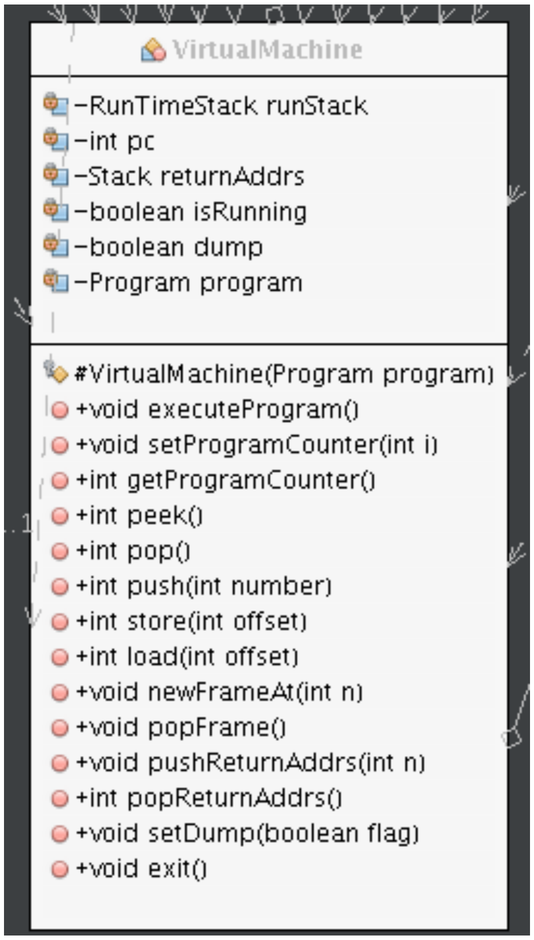
RunTimeStack.java create one arraylist to store each label and a stack call framePointer to store the address which points to the right arraylist element. All the methods are following the requirement from Assignment3.pdf.



```
RunTimeStack
-ArrayList runTimeStack
-Stack<Integer> framePointer

+RunTimeStack()
+void dump()
+int size()
+int peek()
+int pop()
+int push(int i)
+int store(int offset)
+int load(int offset)
+void newFrameAt(int offset)
+void popFrame()
+Integer push(Integer i)
```

ViturалMachine.java create RunTimeStack object called runStack. Then it use runStack to call the same methods in RunTimeStack. This is to keep encapsulation.



```
VirtualMachine
-RunTimeStack runStack
-int pc
-Stack returnAddrs
-boolean isRunning
-boolean dump
-Program program

#VirtualMachine(Program program)
+void executeProgram()
+void setProgramCounter(int i)
+int getProgramCounter()
+int peek()
+int pop()
+int push(int number)
+int store(int offset)
+int load(int offset)
+void newFrameAt(int n)
+void popFrame()
+void pushReturnAddrs(int n)
+int popReturnAddrs()
+void setDump(boolean flag)
+void exit()
```

**<u>Results and conclusion:</u>** When testing fib.x.cod, it takes an integer as input and output the right result. But when test factorial.x.cod, it takes an integer as input, and generate ArrayIndexOfOutBoundException, I thought it has to do with RunTimeStack, but I checked many times that the algorithm of pop method in TunTimesStack is right. Then I used slack to ask for help. Then I found two mistakes, one is at BopCode class that I wrote > instead of < by mistake. After fixing the bug, the output is right, but still generate OutBoundException. I added catch exceptions for  most pop method, I found out that I pop one more then I suppose to pop. Afterwards, both fib and factorial works fine. The most important I learnt from this project is reflection and dynamic binding. It gives us so much power to modify method, classes, argument, etc. The bigger challenge I face is how to use reflection to check each class in order to help resolve address. By goggling online, slack with classmates, and read the slides from class. It helps me understand better about reflection.