CSC 413 Project Documentation

Fall 2018

Ahmad El Shakoushy

ID: 915814671

Assignment #2: Interpreter

CSC 413-01

https://github.com/csc413-02-fa18/csc413-p2-aelshako

Table of Contents

1 Introduction

1.1 Project Overview

The purpose of this project was to design what's known as an interpreter for a programming language known as Language X. Language X is a made-up programming language that we are utilizing in the project. An important point to make is that Language X is not a completely random language; it is actually similar to other, mainstream languages. An interpreter is able to take a simplified form of Language X and perform the needed operations. A good analogy is that of a manufacturing plant where the raw-materials come into the plant in a structured fashion and the plant uses them to make products that suit their clients. In this case, the interpreter would be the manufacturing plant because it takes simplified code in a structured fashion and performs the necessary output. The concept of an interpreter is extremely important in computer science, and its possible uses are showcased via two sample input codes: fib.x.cod and factorial.x.cod. The first file computes the value at a certain index in the legendary Fibonacci sequence while the second computes the factorial of an integer number. In completing this project, I attempted to use what's known as good OOP principles. OOP is simply a way of programming by starting off with blueprints instead of starting from scratch for multiple similiar objects.

1.2 Technical Overview

This section will serve as a brief technical overview of the project. Please visit section 7 of this documentation for a more detailed explanation. The purpose of this project was to design, implement, and verify the functionality of an interpreter while maintaining the use of good OOP principles. The interpreter functions on a programming language known as 'Language X'. Language X can be seen as a simplified form of Java. Our interpreter's duty is to process bytecodes which have already been created from language X source code. The design incorporated various classes and I will give you brief explanation of the functionality of the most important classes as well as how they fit together in the scheme of Computer Science principles.

There are many bytecodes with various functionalities such as pop(pops the RunTime Stack), goto(jumps to a certain label), call(transfers control to the respective function), and args(executed before call to specify the number of arguments to the function). Each specific ByteCode is defined in 'theinterpreter' assignment pdf and thus will not be re-explained here(I've cleared this with the Professor). All of the bytecodes (including the ones I've mentioned) have various common needs/data fields: a method for execution, a method for initialization, and a method to print the state of the bytecode. Naturally, such a case would be approached via the creation of an abstract class. In our case, the abstract class is the ByteCode class which has three abstract methods: init, execute, and toString. These methods are used to implement the operations of the specific bytecode and its respective data fields. A very important point to make is that the subclasses for the abstract ByteCode class should not know more than necessary. In other words, we must adhere to the concept of encapsulation since the individual bytecodes can be seen as the lowest level of the program.

After loading and initialization, bytecodes are executed in a class known as the VirtualMachine. This class maintains the operations of the entire program. Therefore, it includes a program counter to iterate through and execute the lines of the program. An important point to make is that this class includes an instance of a RunTimeStack class. The RunTimeStack class maintains the

FramePointer Stack and the RunTime Stack. For simplicity, we are only allowing for integers to be stored on the RunTime Stack. The FramePointer stack holds integers of the active frames for elements in the RunTimeStack. Encapsulation is being used to ensure that the VirtualMachine sends requests to the RunTimeStack when it requires operations on either of the two previously mentioned stacks. What this boils down to is the fact that the VirtualMachine needs to know what functionalities are available in the RunTimeStack class to be able to request them, but the VirtualMachine does not need to know how they are implemented.

Another important class is the Program class. The program class simply holds the responsibility for storing all of the bytecodes read from the file. It stores these bytecodes into a local array list of bytecodes. There is also a ByteCodeLoader class which is responsible for loading the bytecodes into the Program object. Yet another important class is the CodeTable class which simply holds a static HashMap that is used to map between instruction names and ByteCode subclasses. For example, “HALT” would be mapped to “HaltCode” where “HALT” is the instruction and “HaltCode” is a subclass of ByteCode. This is extremely important, because it is used to make instances of proper objects when reading from the file.

The last class that I’d like to mention is the Interpreter class which serves as the entry point to the program. Interpreter uses the argument to its main function to know which file to open. It then calls on the respective classes for loading bytecodes, initializing bytecodes, and executing the program.

I apologize if this seems very convoluted because the classes have many different functions, and some things are best explained visually as I will do in section 6.1 of this documentation. I am confident that you will have a more in-depth and well-rounded understanding of the project after delving into section 6.

1.3 Summary of Work Completed

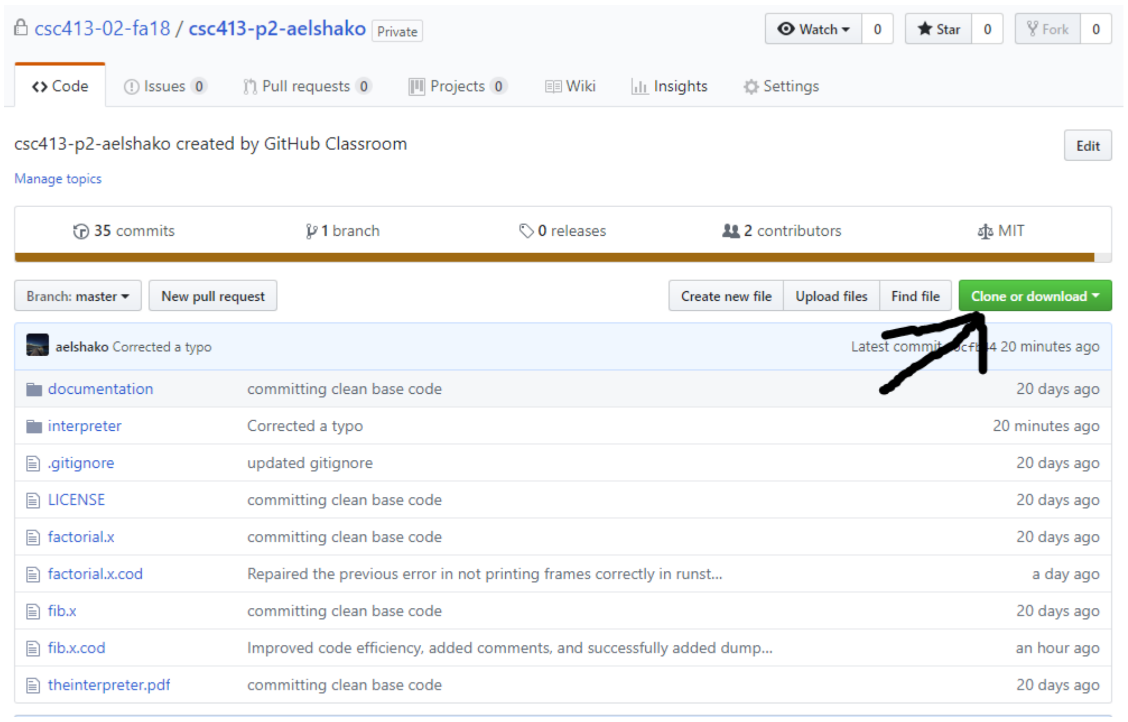
I followed the recommended order of writing the code by first creating all the ByteCode classes but leaving their methods empty. My next step was to implement the ByteCodeLoader, Program, RunTimeStack, and the VirtualMachine classes. My last step was to fill in the empty ByteCode classes and their methods. I have successfully implemented the entire project and feel confident that I’ve followed good Software Engineering principles when possible. I spent countless hours debugging my project, and thus I am confident that it performs all the required functionality flawlessly. I’ve gone through the dumping process line-by-line and verified that my dumping is done correctly for both of the sample files as well as all sample code given in the assignment pdf.

2 Development Environment

a. Version of Java used: Java 10.0.2

b. IDE used: IntelliJ Idea

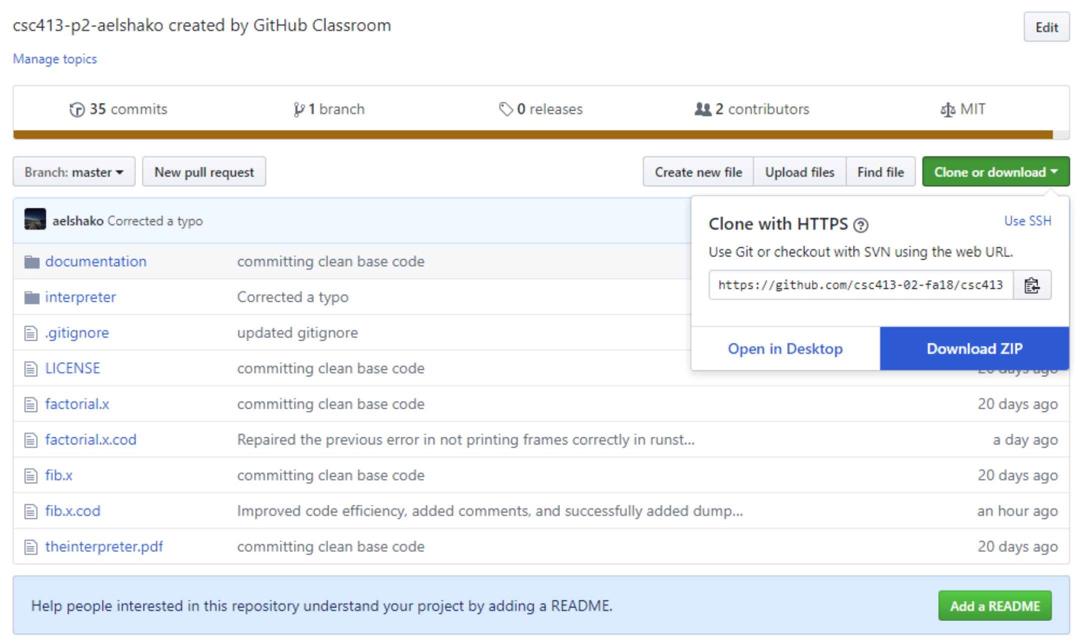
3 How to Build/Import your Project



csc413-p2-aelshako created by GitHub Classroom

aelshako Corrected a typo	Latest commit [illegible]	20 minutes ago
documentation	committing clean base code	20 days ago
interpreter	Corrected a typo	20 minutes ago
.gitignore	updated gitignore	20 days ago
LICENSE	committing clean base code	20 days ago
factorial.x	committing clean base code	20 days ago
factorial.x.cod	Repaired the previous error in not printing frames correctly in runst…	a day ago
fib.x	committing clean base code	20 days ago
fib.x.cod	Improved code efficiency, added comments, and successfully added dump…	an hour ago
theinterpreter.pdf	committing clean base code	20 days ago

Figure(1a): Click the green 'Clone or download' button in the repo link.(The button is pointed to by the arrow in the figure)



csc413-p2-aelshako created by GitHub Classroom

Use Git or checkout with SVN using the web URL.

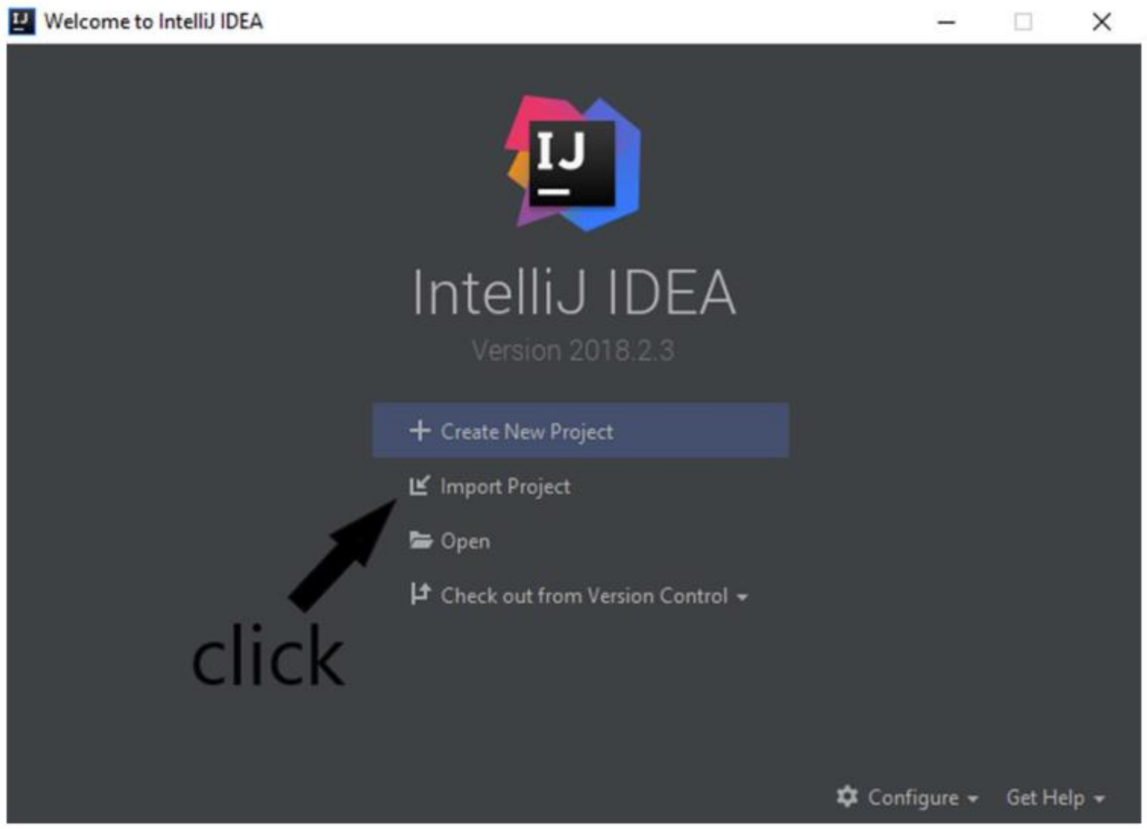
```
https://github.com/csc413-02-fa18/csc413
```

aelshako Corrected a typo		
documentation	committing clean base code	
interpreter	Corrected a typo	
.gitignore	updated gitignore	
LICENSE	committing clean base code	
factorial.x	committing clean base code	20 days ago
factorial.x.cod	Repaired the previous error in not printing frames correctly in runst…	a day ago
fib.x	committing clean base code	20 days ago
fib.x.cod	Improved code efficiency, added comments, and successfully added dump…	an hour ago
theinterpreter.pdf	committing clean base code	20 days ago

Help people interested in this repository understand your project by adding a README.

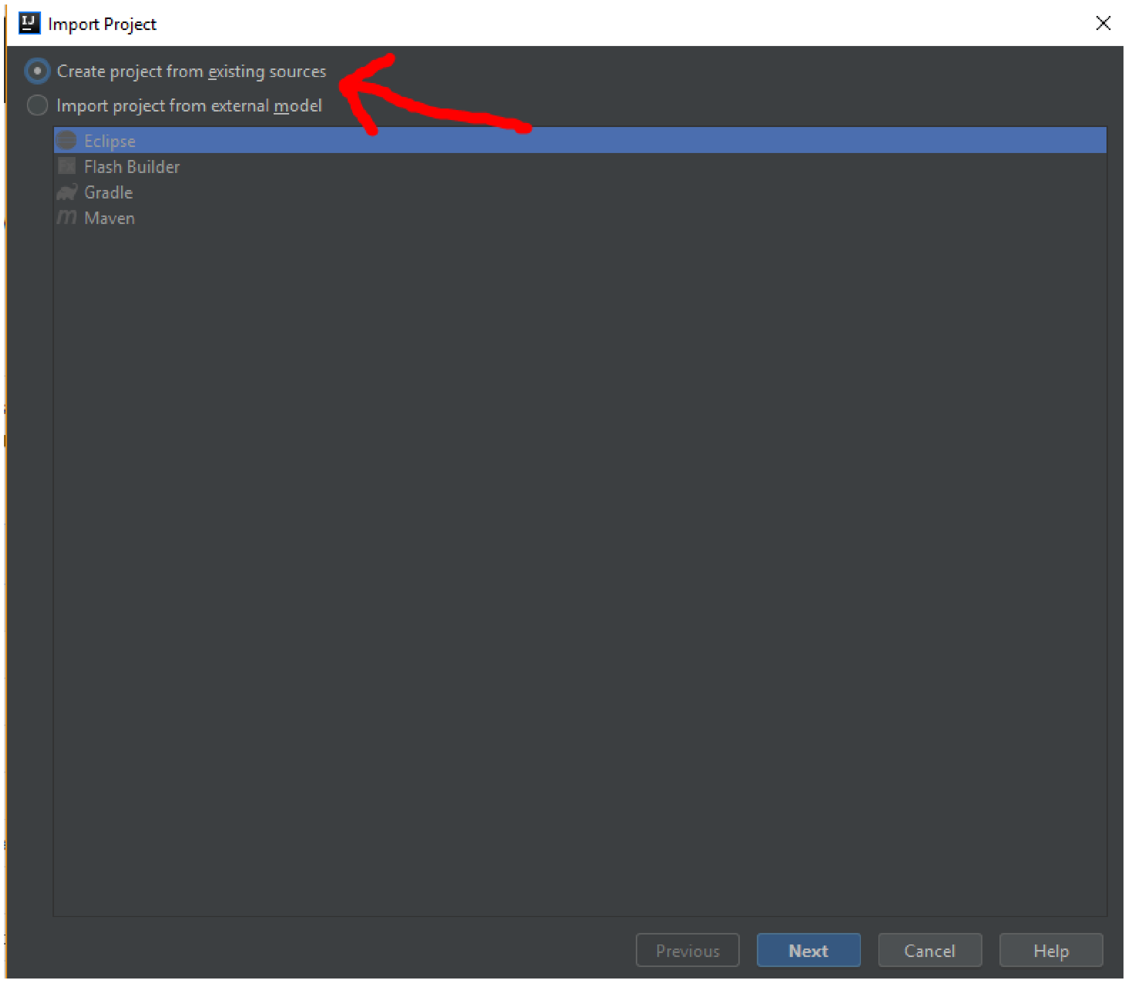
Figure(1b): Click the 'Download ZIP' button. (The highlighted button is blue in the figure)

- Extract the zip folder to a known location

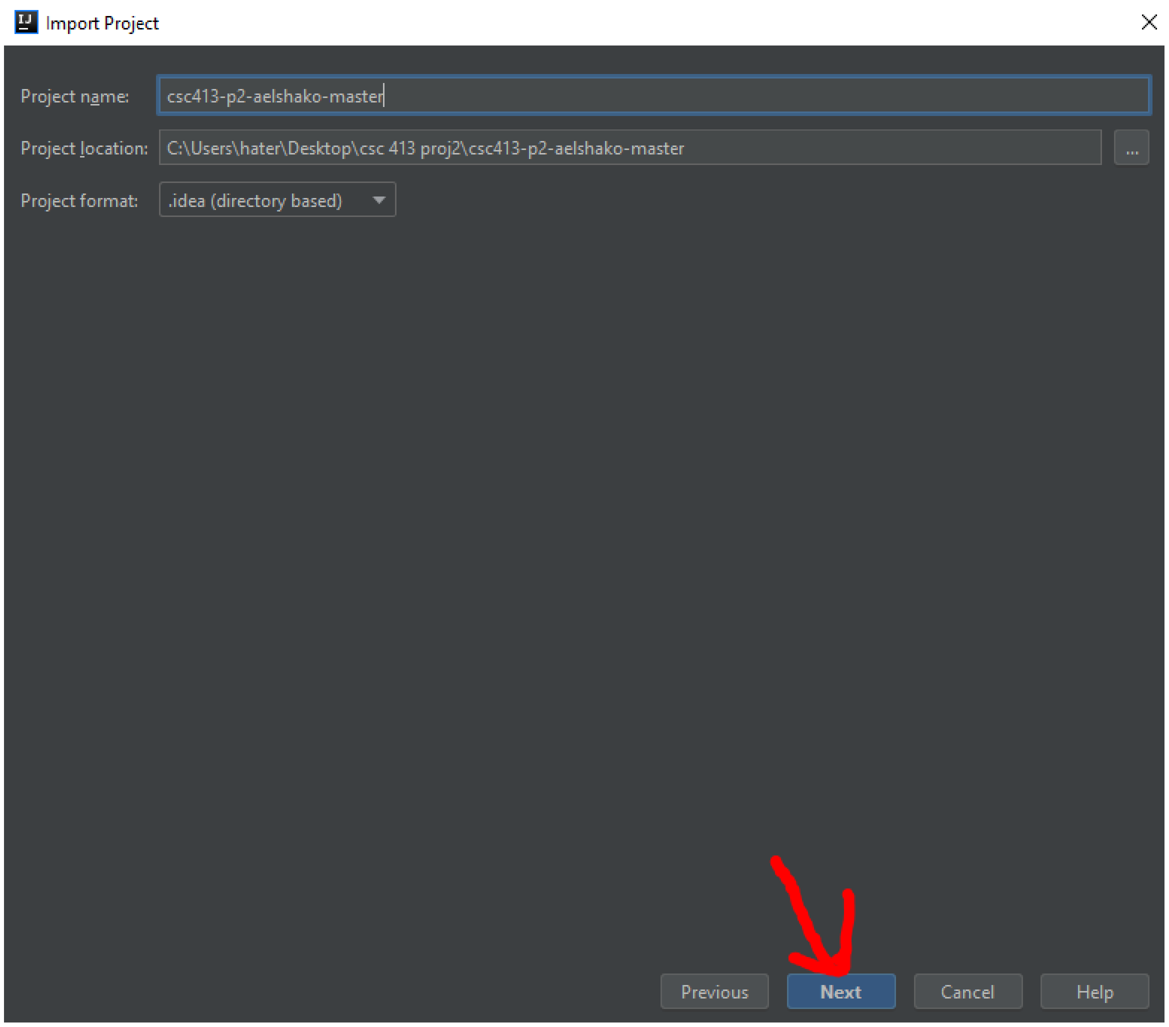


Figure(1c): Click the 'Import Project' button.

- On next screen, select where you extracted the zip file, and press 'OK'.


Import Project
[✓ Create project from existing sources]
Import project from external model
Eclipse
Flash Builder
Gradle
Maven
Previous
Next
Cancel
Help

Figure(1d): Click ‘Create project from existing sources’ and press ‘Next’.

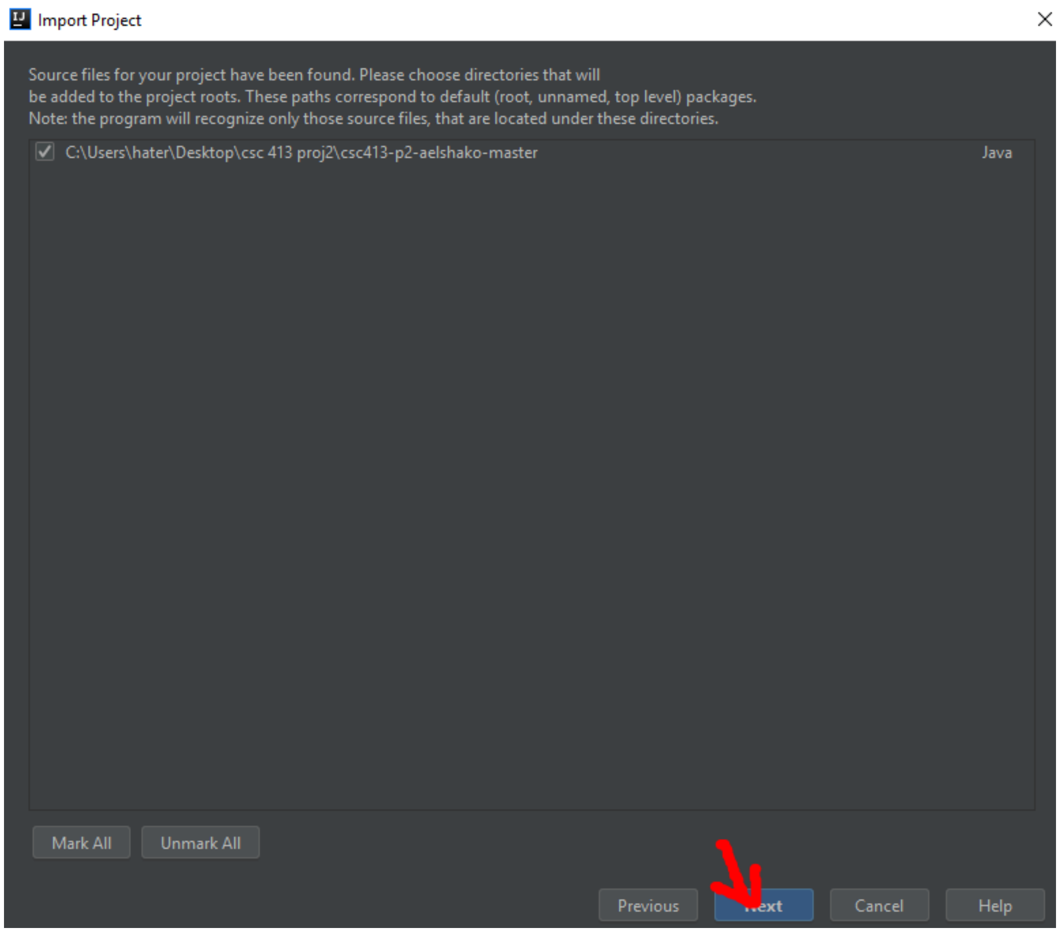

Import Project

Project name: csc413-p2-aelshako-master

Project location: C:\Users\hater\Desktop\csc 413 proj2\csc413-p2-aelshako-master

Project format: .idea (directory based)

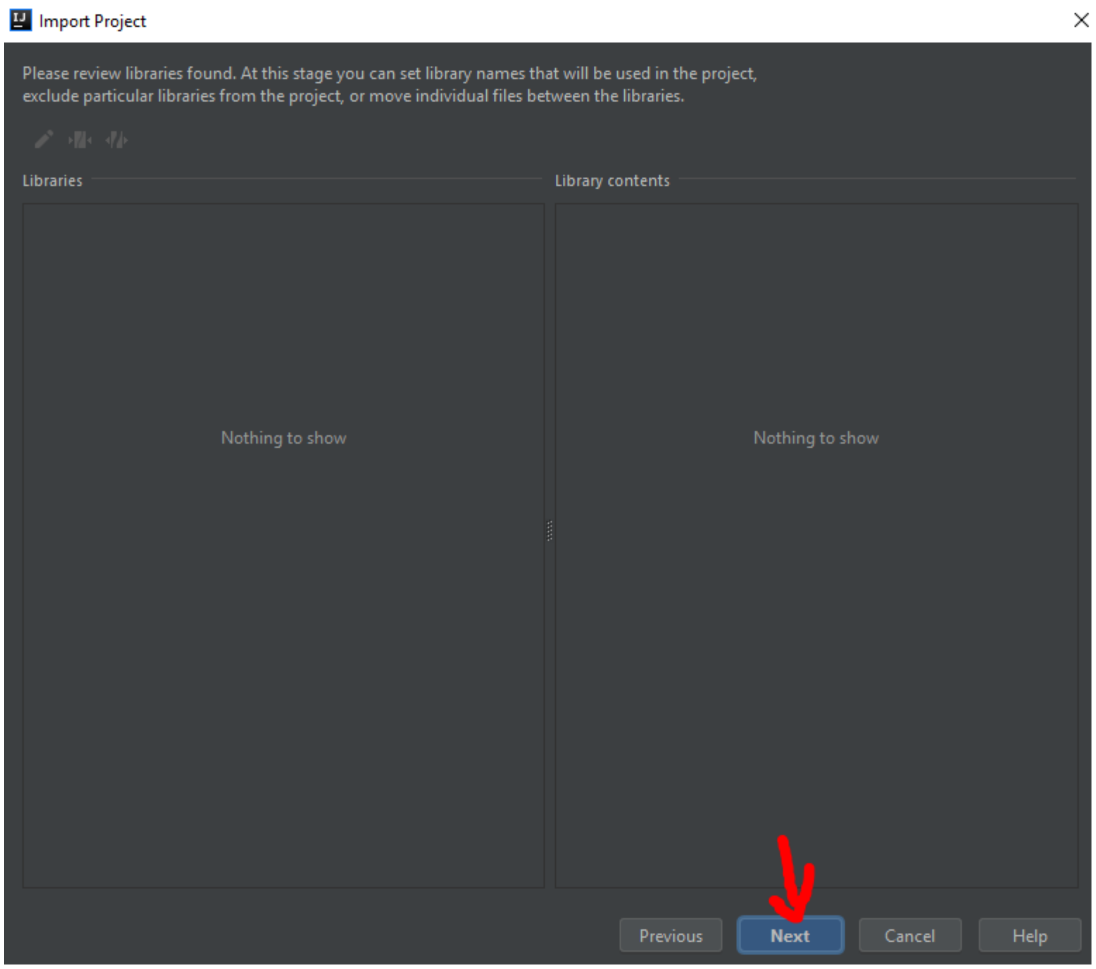
Figure(1e): Verify that the Project format matches the figure and press ‘Next’.


Import Project

Source files for your project have been found. Please choose directories that will
be added to the project roots. These paths correspond to default (root, unnamed, top level) packages.
Note: the program will recognize only those source files, that are located under these directories.

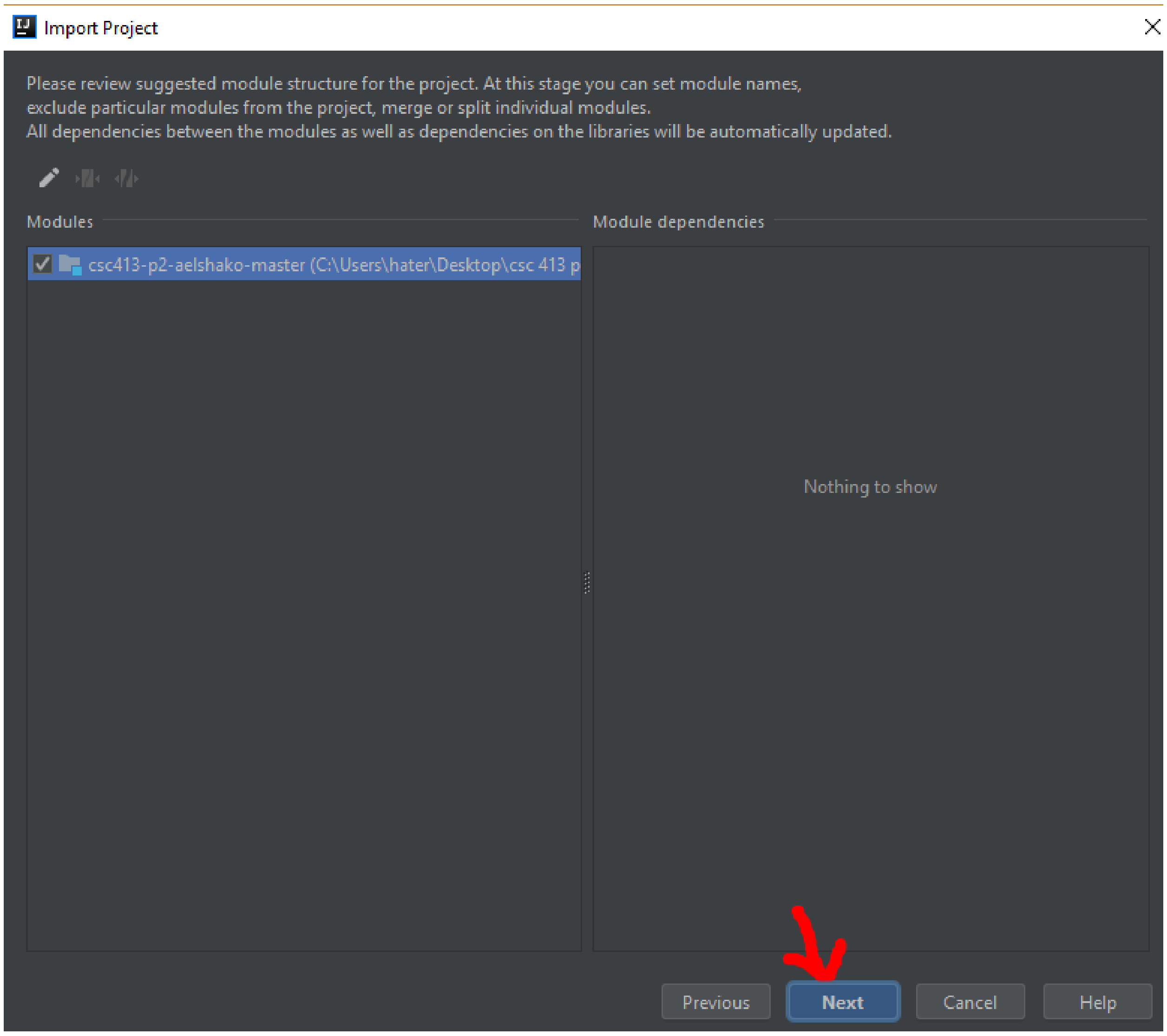
✓ C:\Users\hater\Desktop\csc 413 proj2\csc413-p2-aelshako-master Java

Figure(1f): Click ‘Next’.


Import Project

Please review libraries found. At this stage you can set library names that will be used in the project, exclude particular libraries from the project, or move individual files between the libraries.

Figure(1g): Click ‘Next’.



Import Project

Please review suggested module structure for the project. At this stage you can set module names, exclude particular modules from the project, merge or split individual modules.
All dependencies between the modules as well as dependencies on the libraries will be automatically updated.

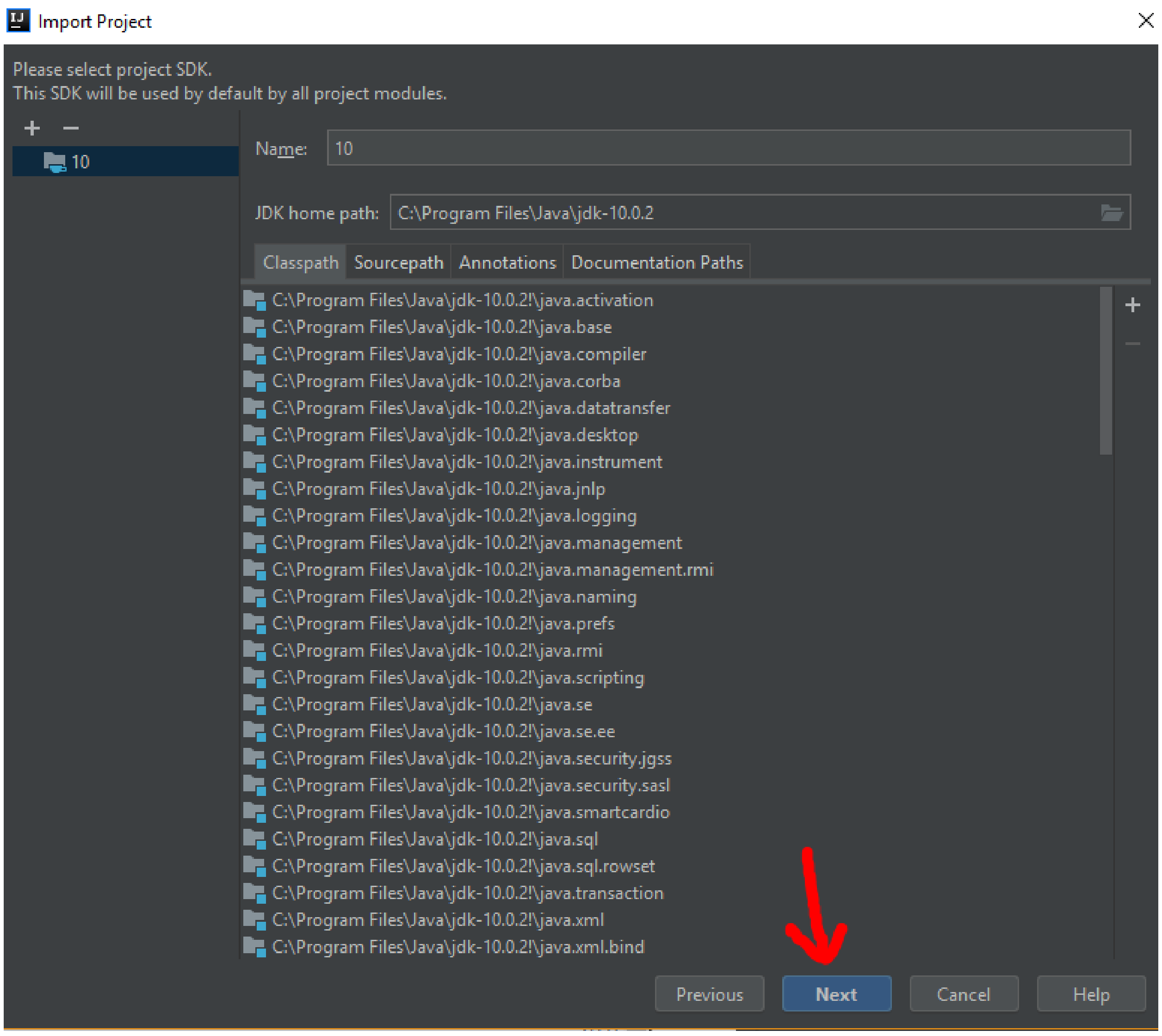
Modules

☑ csc413-p2-aelshako-master (C:\Users\hater\Desktop\csc 413 p

Module dependencies

Nothing to show

Figure(1h): Click ‘Next’.



Import Project

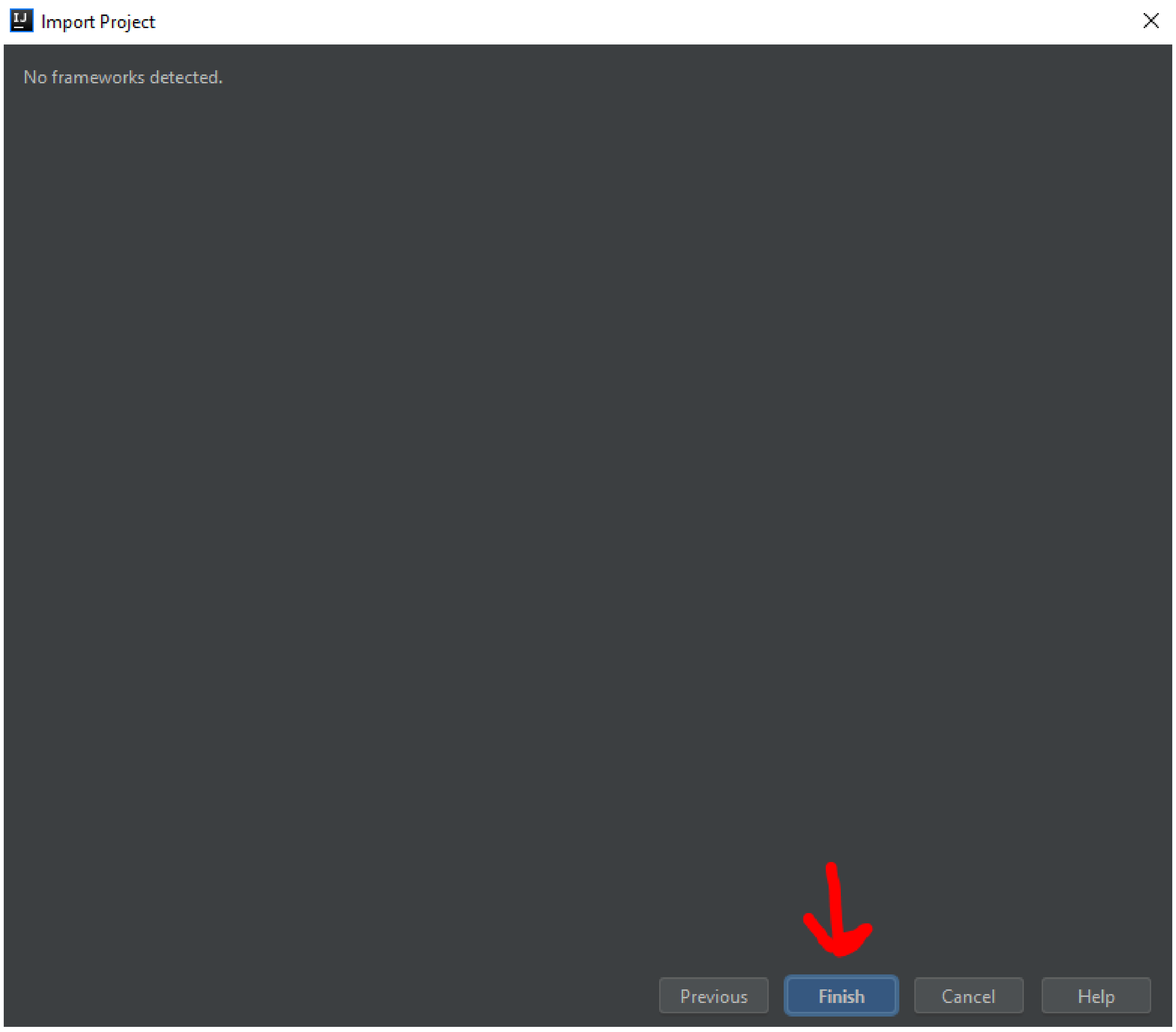
Please select project SDK.
This SDK will be used by default by all project modules.

Name: 10

JDK home path: C:\Program Files\Java\jdk-10.0.2

```
C:\Program Files\Java\jdk-10.0.2!\java.activation
C:\Program Files\Java\jdk-10.0.2!\java.base
C:\Program Files\Java\jdk-10.0.2!\java.compiler
C:\Program Files\Java\jdk-10.0.2!\java.corba
C:\Program Files\Java\jdk-10.0.2!\java.datatransfer
C:\Program Files\Java\jdk-10.0.2!\java.desktop
C:\Program Files\Java\jdk-10.0.2!\java.instrument
C:\Program Files\Java\jdk-10.0.2!\java.jnlp
C:\Program Files\Java\jdk-10.0.2!\java.logging
C:\Program Files\Java\jdk-10.0.2!\java.management
C:\Program Files\Java\jdk-10.0.2!\java.management.rmi
C:\Program Files\Java\jdk-10.0.2!\java.naming
C:\Program Files\Java\jdk-10.0.2!\java.prefs
C:\Program Files\Java\jdk-10.0.2!\java.rmi
C:\Program Files\Java\jdk-10.0.2!\java.scripting
C:\Program Files\Java\jdk-10.0.2!\java.se
C:\Program Files\Java\jdk-10.0.2!\java.se.ee
C:\Program Files\Java\jdk-10.0.2!\java.security.jgss
C:\Program Files\Java\jdk-10.0.2!\java.security.sasl
C:\Program Files\Java\jdk-10.0.2!\java.smartcardio
C:\Program Files\Java\jdk-10.0.2!\java.sql
C:\Program Files\Java\jdk-10.0.2!\java.sql.rowset
C:\Program Files\Java\jdk-10.0.2!\java.transaction
C:\Program Files\Java\jdk-10.0.2!\java.xml
C:\Program Files\Java\jdk-10.0.2!\java.xml.bind
```

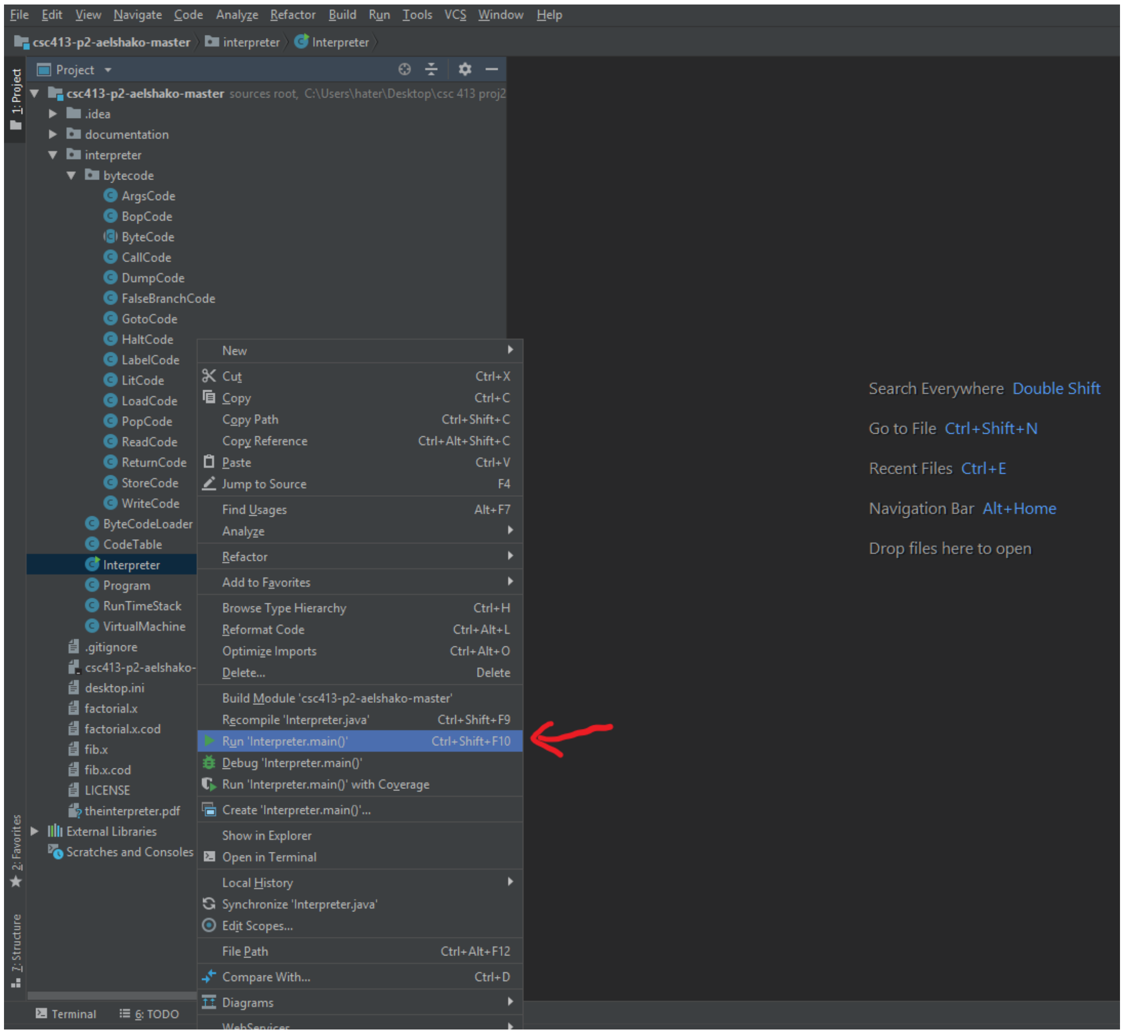
Figure(i): Click ‘Next’.



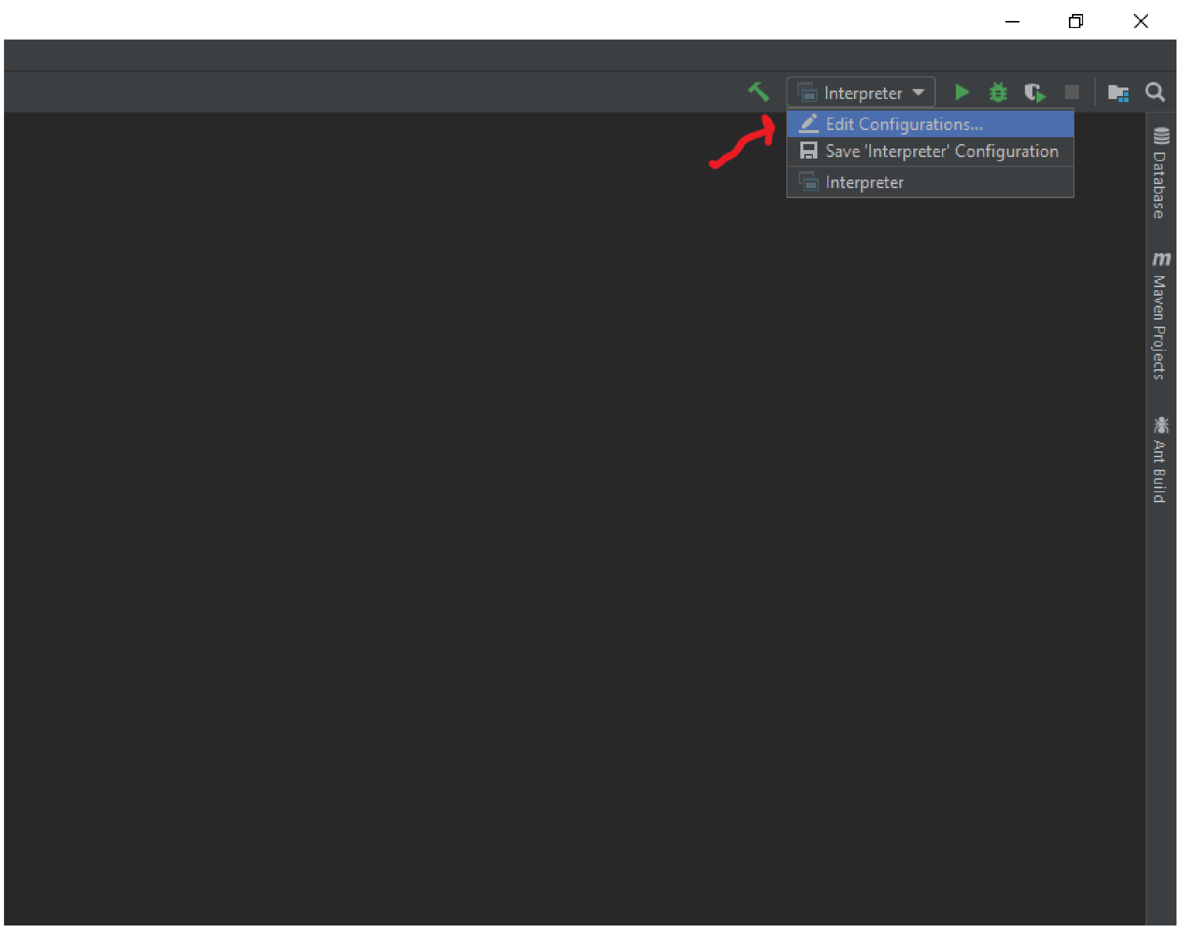
No frameworks detected.

Figure(i): Click ‘Finish’.

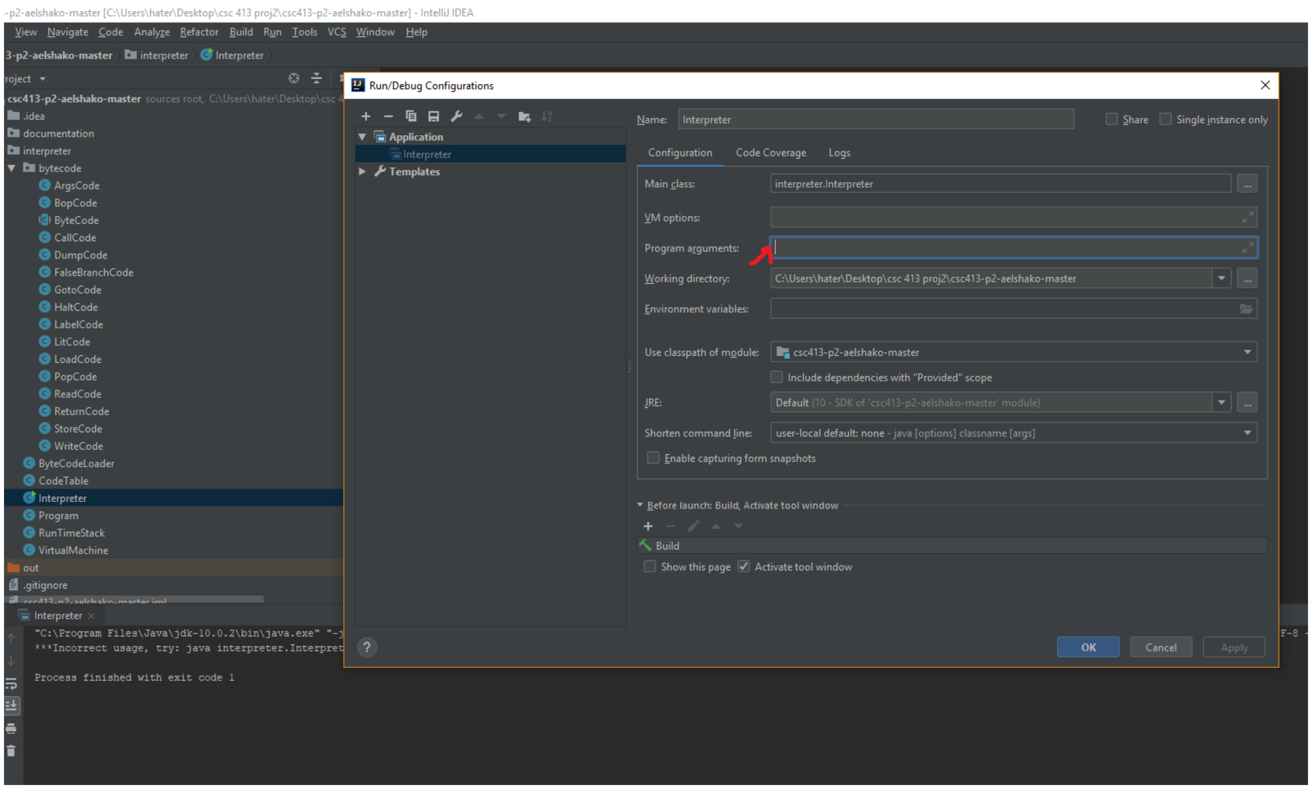
4 How to Run your Project



Figure(2a): Right click on ‘interpreter’ in the left pane and click ‘Run interpreter.main()’. This step is necessary so we can set the configuration in the next step.



Figure(2b): Go to the upper right area of the IntelliJ IDEA window and click on ‘Interpreter’ and then ‘Edit Configurations…’.


Name: Interpreter

Main class: interpreter.Interpreter

Program arguments:

Working directory: C:\Users\hater\Desktop\csc 413 proj2\csc413-p2-aelshako-master

Use classpath of module: csc413-p2-aelshako-master

JRE: Default (10 - SDK of 'csc413-p2-aelshako-master' module)

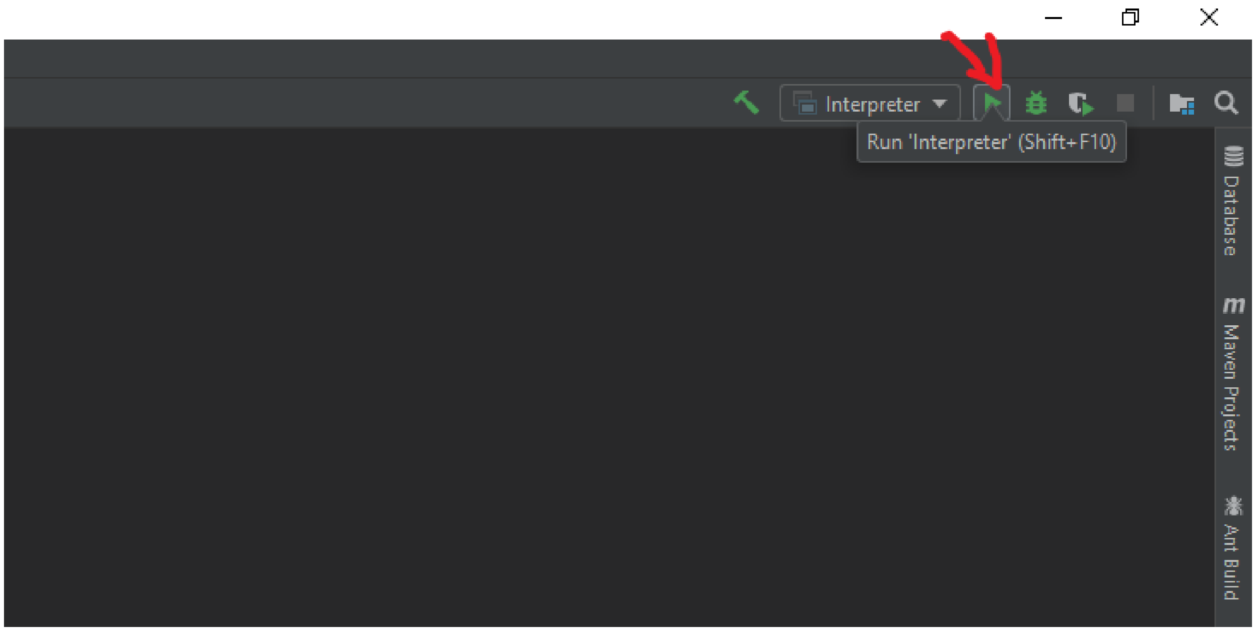
Shorten command line: user-local default: none - java [options] classname [args]

Before launch: Build, Activate tool window

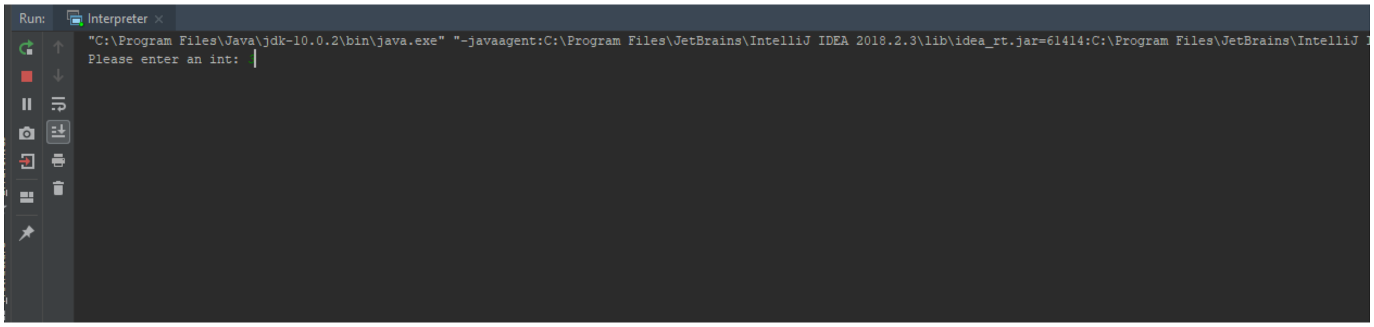
```
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe" "-j
***Incorrect usage, try: java interpreter.Interpret

Process finished with exit code 1
```

Figure(2c): Go to the 'Program arguments' field and type in the name of the program and click 'OK'. I will write in 'factorial.x.cod' without the single quotes.

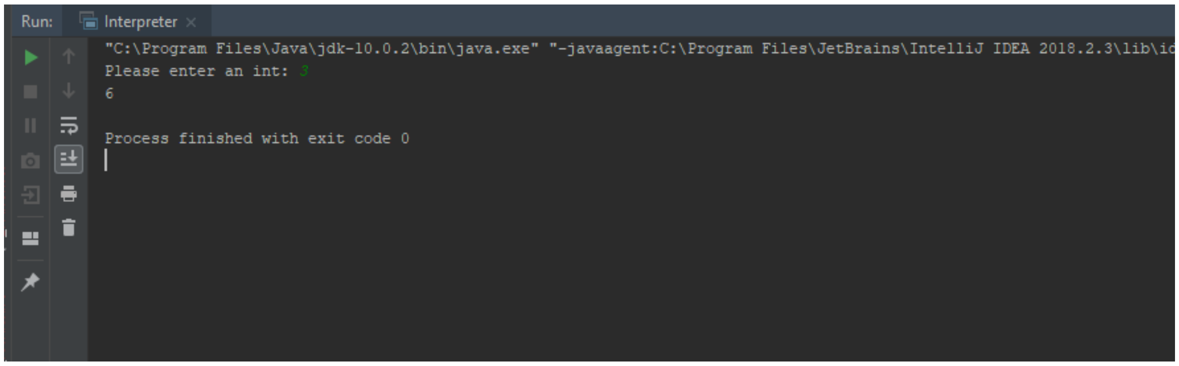


Figure(2d): Click 'Run 'Interpreter''. And your output should show up as shown in the next figure.



```
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2018.2.3\lib\idea_rt.jar=61414:C:\Program Files\JetBrains\IntelliJ I
Please enter an int: 3
```

Figure(2e): You can now enter an integer as prompted by the program. (Make sure to press the 'ENTER' key after entering the integer).



```
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2018.2.3\lib\id
Please enter an int: 3
6

Process finished with exit code 0
```

Figure(2f): The output should show up on the console at shown in the figure. (Note: Each instruction would be 'dumped' and shown on the terminal if a 'DUMP ON' instruction was placed anywhere in the source code.)

5 Assumption Made

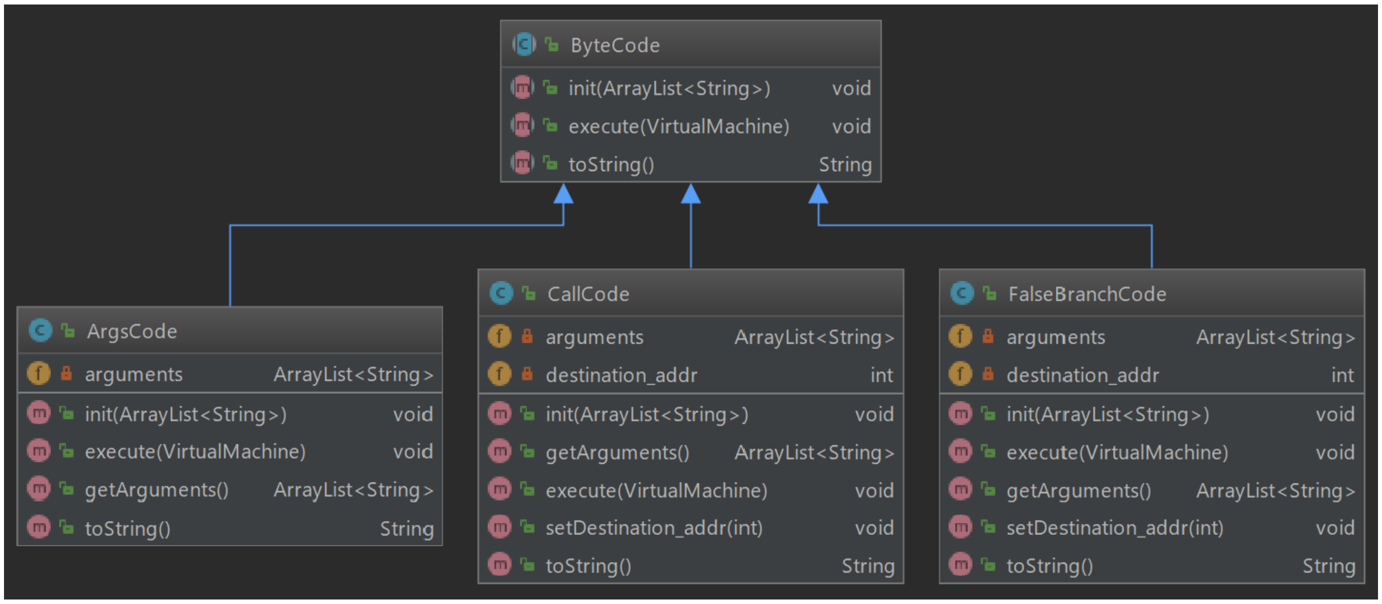
The first assumption that I made was that the number of arguments in the source code are valid. For example, I assume that a return statement for the source code will either be in the form of "RETURN" or in the form of "RETURN f<<2>>" or even in the form "RETURN c". For the "RETURN c" case, I am assuming that the base id is c. I was initially assuming that user input was valid and that it was an integer for the READ ByteCode. However, I later improved my implementation to continue prompting the user until a valid input is entered. I attempted as much I could to account for a bytecode having the wrong number of arguments but I can't guarantee that my program will function and/or output properly if given the wrong number of arguments. However, I can guarantee with the highest degree of certainty that my program will not crash or throw exceptions in such cases. The reason behind this is because I've strategically placed several try-catches to catch any exceptions that may occur due to an incorrect number of arguments.

6 Implementation Discussion

I feel that the implementation process and class hierarchy for this project is best explained visually. Therefore, I will show detailed class diagrams for each unique class in the project and explain its operations. I will start at the lowest level with the ByteCode class as I feel that this makes it easier to follow.

6.1 Class Diagrams(w/ explanations)

ByteCode class and (some) of its subclasses:



ByteCode
init(ArrayList<String>) void
execute(VirtualMachine) void
toString() String
ArgsCode
arguments ArrayList<String>
init(ArrayList<String>) void
execute(VirtualMachine) void
getArguments() ArrayList<String>
toString() String
CallCode
arguments ArrayList<String>
destination_addr int
init(ArrayList<String>) void
getArguments() ArrayList<String>
execute(VirtualMachine) void
setDestination_addr(int) void
toString() String
FalseBranchCode
arguments ArrayList<String>
destination_addr int
init(ArrayList<String>) void
execute(VirtualMachine) void
getArguments() ArrayList<String>
setDestination_addr(int) void
toString() String

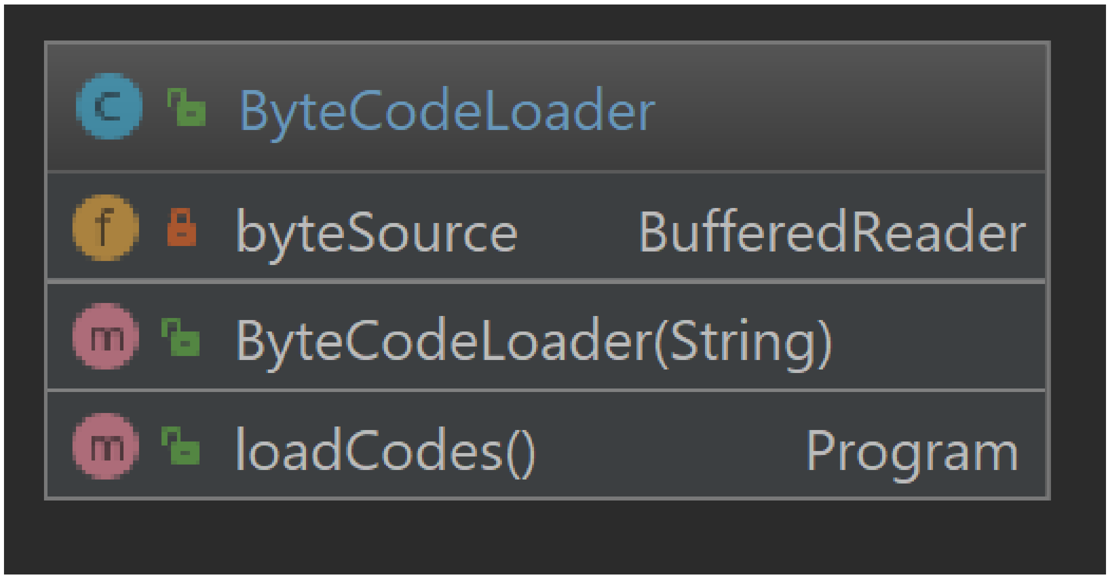
Figure (3a): This figure shows the structure of the abstract ByteCode class and 3 of its subclasses(ArgsCode, CallCode, and FalseBranchCode). I'm only showing 3 subclasses because all of the subclasses are basically identical with the exception of having a destination_addr variable and a setter for that variable in the branching bytecodes(CallCode, GotoCode, FalseBranchCode). The left-most subclass in the figure shows the structure of the non-branching subclasses. There is a total of 15 ByteCode subclasses(ArgsCode, BopCode, CallCode, DumpCode, FalseBranchCode, GotoCode, HaltCode, LabelCode, LitCode, LoadCode, PopCode, ReadCode, ReturnCode, StoreCode, and WriteCode). I will not explain each class as they are already explained in a very thorough fashion in 'theinterpreter' assignment pdf.

The ByteCode class is an abstract class because each separate bytecode requires an init method, an execute method, and a toString method to operate. The abstract ByteCode class has a public initialization fuction(init(ArrayList<String>)) which is implemented in all of its subclasses. The purpose of this method is to take in an ArrayList of Strings and to assign it to the local private arguments variable in the subclasses. Each ByteCode subclass has a public getArguments() method to return the arguments of the ByteCode. The second method in the abstract ByteCode class which is implemented by its subclasses is the execute method which takes in an object of the VirtualMachine. Essentially, the VirtualMachine passes itself into the execute method of each ByteCode when its time for their execution. Having this level of separation is crucial in the program as it ensures that the ByteCodes do not have direct access to the RunTime stack or any of its operations. All ByteCode operations must go through the VirtualMachine class or this would break encapsulation. The execute method performs many different functionalities depending on the specific ByteCode being executed. For example, many checks/operations are done in the execute method of BopCode while no operations performed in the execute method of LabelCode.

The third and last method listed in the abstract ByteCode class and implemented in all its subclasses is the toString() method. This method is used for dumping the state of that particular ByteCode when dumping is enabled in the VirtualMachine class. I will go into more detail on dumping in the VirtualMachine class section of this documentation.

As mentioned previously, the ByteCodes that are used in order to jump to different labels in the program(GotoCode, FalseBranchCode, and CallCode) need an extra private field in order to hold the destination(line number) address of where they should jump to. This field requires a public method (setDestination_addr(int dest)) to set the private local destination variable. An important fact that I'd like to bring to your attention is that the local destination address variable has a setter method but does not have a getter method because it is only used within its class, and thus should never be accessed outside. In order to jump to a Label, the jumping ByteCodes set the PC(program counter) to the address of where they should jump among other required functionalities as defined in the given specification pdf. I've offset the PC(program counter) value by -1 to ensure that the labels which are being jumped to get executed, so we are able to print them in the case where dumping is enabled.

ByteCodeLoader

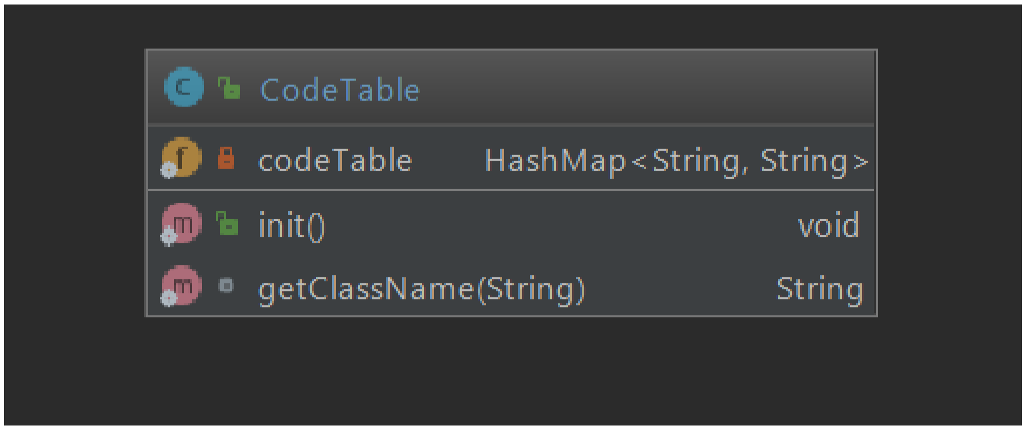


```
ByteCodeLoader
byteSource                 BufferedReader
ByteCodeLoader(String)
loadCodes()                Program
```

Figure(3b): This figure shows the structure of the ByteCodeLoader class. The skeleton code given to use for this class was explicitly extending the Object class which I found to be rather odd since every class in Java implicitly extends the Object class. Therefore, I removed the explicit extension of the Object class to reduce redundancy in my code. The purpose of the ByteCodeLoader class is to initialize the ByteCodes via their init method and to add them to an ArrayList within the Program class. I will go into more detail regarding the Program class in the Program class section of this documentation. The constructor for this class takes in a String representing the file name.

The ByteCodeLoader performs all of its functionality in its loadCodes() method by using its local BufferedReader object: byteSource. In the ByteCodeLoader, we go through the source code file line-by-line and make an instance of the appropriate ByteCode subclass based on its name. We use the CodeTable class to map the instruction in the source code with that of the actual ByteCode subclass. I will explain the CodeTable class in more detail, but it is enough to understand that it a "RETURN" in the source code file would be mapped to “ReturnCode” where “ReturnCode” is the proper class name. After making an instance of the proper class, we use a StringTokenizer to iterate over the all the arguments for the given ByteCode. We then use the init method to initialize the local arguments variable for that specific ByteCode subclass object. The last step, as mentioned previously is to add the code into an ArrayList within an object of the Program class. The fully initialized Program object which holds all of our initialized ByteCodes is then returned at the end of the LoadCodes() method.

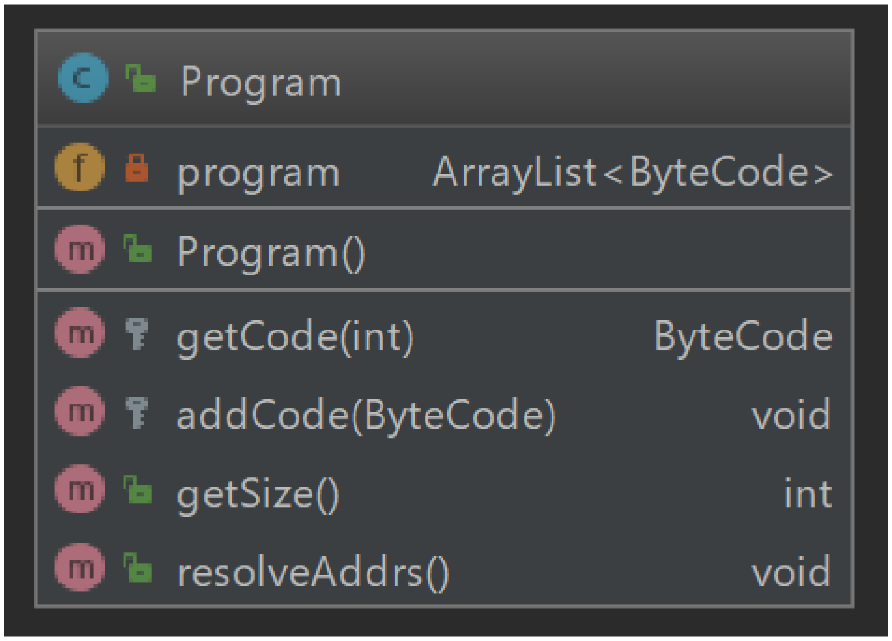
<u>CodeTable</u>



```
CodeTable
codeTable              HashMap<String, String>
init()                                    void
getClassName(String)                    String
```

Figure(3c): This figure shows the structure of the CodeTable class. The CodeTable class is a very simple class which holds a private static HashMap of type<String, String> to map between keys(instruction names) and the names of the actual ByteCode subclasses which corresponds to those instructions. As mentioned in the previous section, the CodeTable class is used in the ByteCodeLoader in order to allow for the creation of the appropriate ByteCode subclass instances. Specifically, the static getClassName method is used in order to retrieve elements from the private HashMap. The reason that the methods and the local field of the codeTable class are all static is because they belong to the actual class itself as opposed to instances of the class. This makes sense because the values in the HashMap are not changed and it would be very inefficient for them to be re-initialized each time.

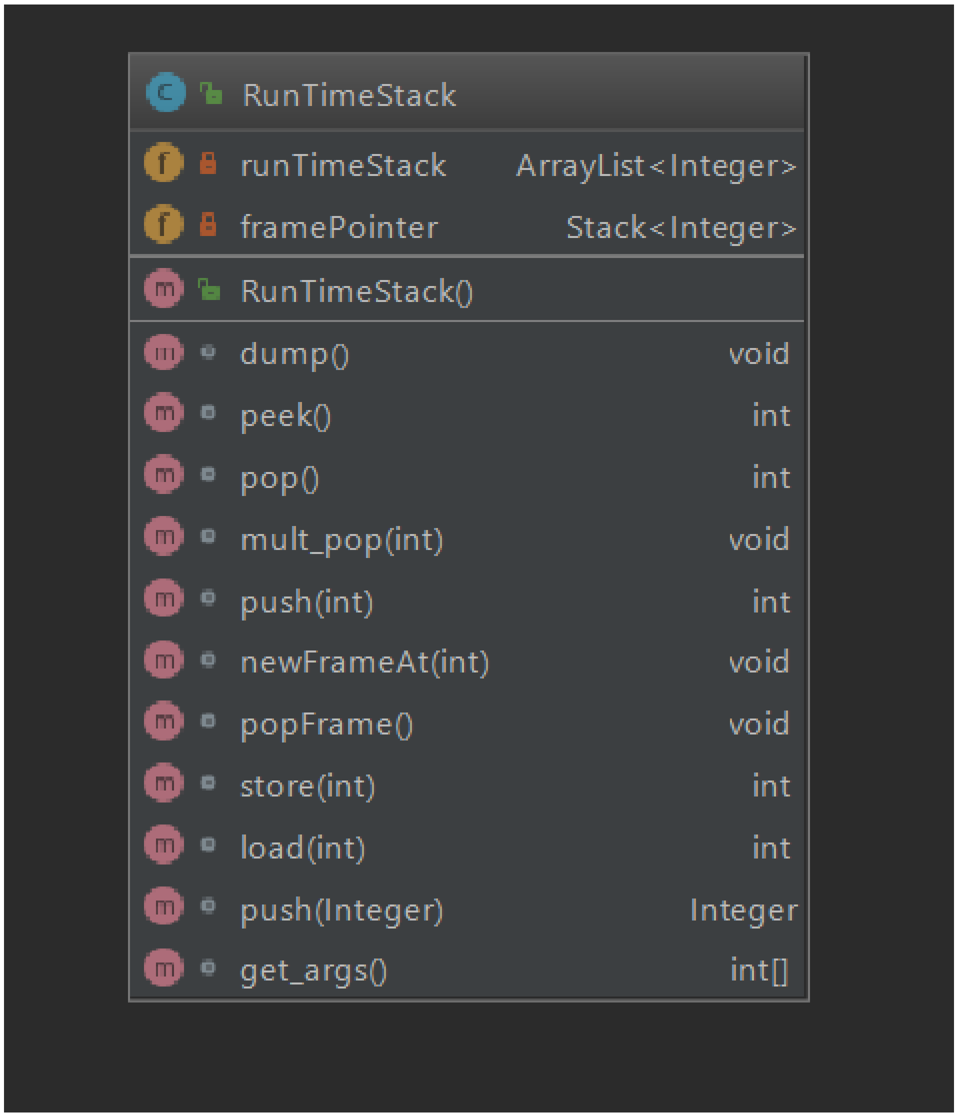
<u>Program</u>



Figure(3d): This figure shows the program class which is responsible for the storage of all the ByteCodes read from the source code file as well as address resolution. The class has a private local ArrayList of type<ByteCode> to hold the ByteCodes. We are explicitly listing the type of the ArrayList as <ByteCode> to ensure that it can only hold ByteCodes. The Program() constructor simply allocated memory on the heap for our program ArrayList. There is a public getSize() method within the class to return the size of the program ArrayList within a given Program object. There are two protected methods within the class: getCode(int) and addCode(ByteCode). I've set the visibility of these methods to protected because they are only called in the same package(interpreter). The getCode method takes in an integer(the PC value) and returns the ByteCode within the program ArrayList at that index. This essentially allows you to retrieve ByteCodes that are on a certain line of your program. The last and arguably the most important method for this class is the resolveAddrs() method. In resolveAddrs(), we use a HashMap of type<String, Integer> in order to map labels to the line numbers that they occur at. Essentially, we loop through all the ByteCodes once and identify any LabelCodes. If LabelCodes are found, their label is added to our HashMap and their line number is added to our HashMap as an Integer.

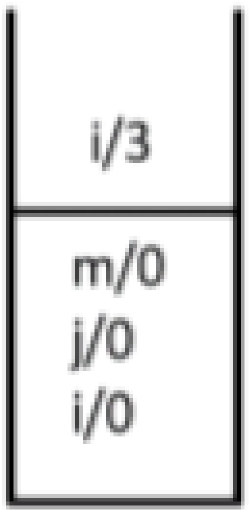
The next step is to loop through all the ByteCodes again in order to look for jumping ByteCodes(FalseBranchCode, GotoCode, CallCode) via the utilization of the instanceof operator. If any of the aforementioned ByteCodes are found; we retrieve their line number from the previously initialized HashMap and use the setDestination_addr method in that specific ByteCode subclass to set the destination address.

RunTimeStack



RunTimeStack	
runTimeStack	ArrayList<Integer>
framePointer	Stack<Integer>
RunTimeStack()	
dump()	void
peek()	int
pop()	int
mult_pop(int)	void
push(int)	int
newFrameAt(int)	void
popFrame()	void
store(int)	int
load(int)	int
push(Integer)	Integer
get_args()	int[]

Figure(3e): This figure shows the structure of the RunTimeStack class. The RunTimeStack class is responsible for maintaining the active frames. It contains two very important private data fields: a runTimeStack of type(ArrayList<Integer>) and a framePointer of type(Stack<Integer>). We are using an ArrayList for the runTimeStack variable, because we will need to access all the indices within the runtime stack. I've restricted both of the values held by these data structures to be of the Integer object type to ensure that our data structures hold the proper type. The framePointer stack holds the indices of the first element in each frame. The top element on the framePointer stack would be the index(from bottom) of the first element in that frame within the runtime stack. Consider the following snapshot of a runtime stack from the given assignment pdf:



Figure(3f): This figure shows an example of the runtime stack at a given moment. Please note that the letters on the left and the slash symbol are not actually stored on the stack. The stack simply holds the integers shown on the write. The framepointer stack for this runtime stack would hold 0(in the bottom) and 3 on top because those are the indices of the first element in each frame. The drawn horizontal line is a representation of a frame divider but doesn't refer to anything special being stored on the runtime stack.

As you've probably noticed, the runtime stack is intertwined with the frame pointer stack. Therefore, we will often utilize both stacks when implementing functionalities to perform certain operations on one or the other. The dump function in the class is used for dumping the runtime stack when the dumping flag is set. For example, it would print something like: [0,3,4] [6][][2] where the brackets represent distinct frames and the integers represent values of the runtime stack. Please note that the dumped runtime stack grows from left to right. This simply means that the top of the stack is the right-most element in the output. All of the methods within the runtime stack explained below this point have package private visibility as they do not need to be called outside of the class in any feasible scenario.

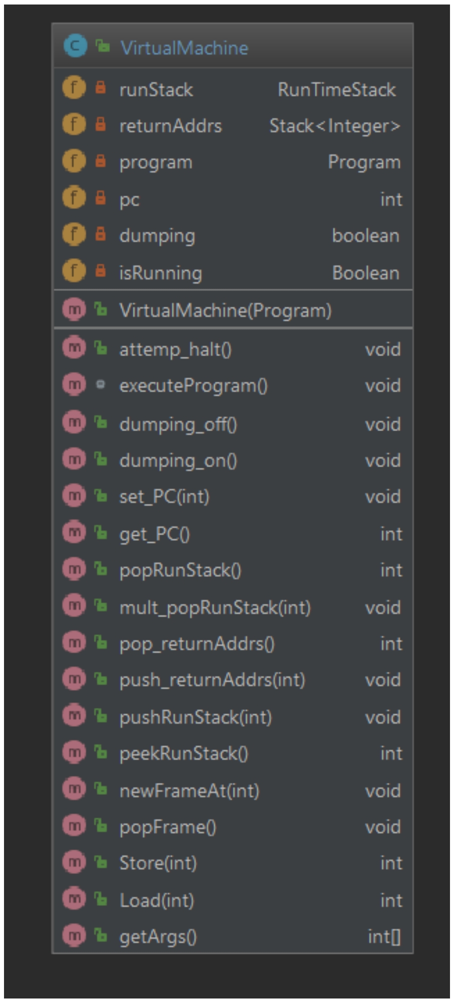
The peek() method of the class simply returns the value at the top of the runtime stack. The pop method will pop the runtime stack and return that value if and only if there are enough elements in the current frame. In pop(), we simply do nothing and return 0 if the frame is empty. The mult_pop(int num_to_pop) method is a helper method that I've created to pop a given number of elements from the runtime stack when we don't need their values. Please note that I don't do any error checking in the mult_pop method because I handle that error checking before I call the num_to_pop method. Num_to_pop is very useful for things like popping frames because it makes the code easier to read and understand. The push(int push_val) method allows for pushing a value onto the runtime stack as well as returning that value. There is an additional push method to push an Integer onto the runtime stack. I personally feel that it would realistically be unnecessary to implement both the int and and Integer version of the push method, but I've implemented it since it's a requirement in the assignment pdf.

The newFrameAt(int offset) takes in an offset which denoted how many values to descend from the top of the runtime stack before making a new frame. The popFrame() method will simply pop the top-most frame via the utilization of the num_to_pop method.

The store(int offset) method takes in an integer offset which denotes how high from the bottom of the runtime stack to transfer the top-most element on the stack. The most important thing about the store method is that it results in the runtime stack being 1 element shorter than it was previously. The load(int offset) method takes in integer offset which denotes the offset from the bottom of the frame to copy a value from before pushing that value onto the stack. The most important thing about the load method is that it results in the runtime stack being 1 element longer than it was previously.

The last method in this class is the get_args() method which loops through the current frame to get and store the arguments after a CallCode is executed with dumping enabled. This allows us to print the arguments outside of CallCode's toString() because we should not be passing things from the RunTimeStack class directly to the ByteCode subclasses to prevent from breaking encapsulation.

<u>VirtualMachine</u>



VirtualMachine

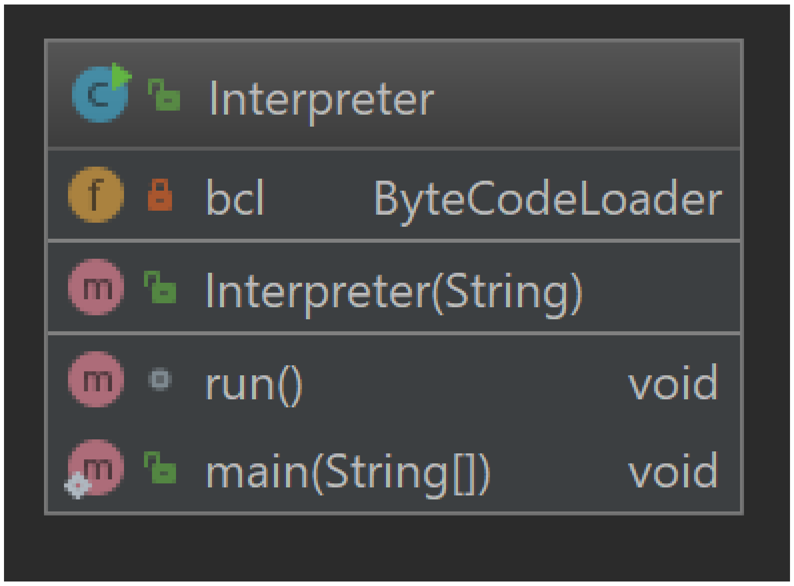
runStack	RunTimeStack
returnAddrs	Stack<Integer>
program	Program
pc	int
dumping	boolean
isRunning	Boolean
VirtualMachine(Program)	
attemp_halt()	void
executeProgram()	void
dumping_off()	void
dumping_on()	void
set_PC(int)	void
get_PC()	int
popRunStack()	int
mult_popRunStack(int)	void
pop_returnAddrs()	int
push_returnAddrs(int)	void
pushRunStack(int)	void
peekRunStack()	int
newFrameAt(int)	void
popFrame()	void
Store(int)	int
Load(int)	int
getArgs()	int[]

Figure(3g): This figure shows the structure of the VirtualMachine class. The VirtualMachine class serves as the controller of the program. The RunTimeStack class is sent requests by the VirtualMachine when it needs certain operations that are not within its respective scope of operation. As you can see in the figure, the RunTimeStack has many methods, and a significant amount of them are identical to those in the RunTimeStack class. All methods referring to the runtime stack or to the FramePointer stack are simply forwarding requests to the previously explained methods in the RunTimeStack class and thus will not be re-explained. For example, the popRunStack() method simply forwards a call to the appropriate method within the RunTimeStack class to pop the runtime stack. Note that the VirtualMachine class directly pop the runtime stack as that would break encapsulation within our program. The constructor of the VirtualMachine class takes in a Program object and uses it to set its local Program object data field. This allows for accessing the ByteCodes within the VirtualMachine class.

The VirtualMachine class holds a RunTimeStack object, a returnAddrs stack to hold Integer return addresses for all functions, a program counter(pc), an isRunning Boolean variable to know if the program has been halted, a Program object program which is a reference to the data structure that holds all the ByteCodes, and a dumping boolean variable to hold the dumping flag. All of these data fields have private accessibility.

The VirtualMachine class has public setters and getters for the pc variable and for the dumping variable. Arguably, the most important function in the VirtualMachine class is the executeProgram() method which increments the pc and continues executing ByteCodes by calling their respective execute methods for as long as isRunning is set to true. This method also checks for the dump flag's status and calls upon each ByteCode's toString to dump that ByteCode. It also dumps the runtime stack after each instruction is executed if dumping is enabled. This method is where I append to the output of my toString() methods for the special cases where I must show arguments or other values from the runtime stack while dumping.

<u>Interpreter</u>



Figure(3h): This figure shows the class structure of the Interpreter class which serves as the entry point to the entire project. The Interpreter has a constructor in which it takes a String that represents the name of the source code file. In the constructor, the CodeTable gets initialized and the private local ByteCodeLoader variable is initialized. The run() method creates a Program instance using the return value of the loadCodes() method in the local ByteCodeLoader variable. The run() method also creates an instance of the virtual machine using the initialized Program object, and finally invokes the executeProgram() method in the VirtualMachine to execute the ByteCodes. The main method's arguments are used to invoke the run() method with the appropriate file name .

7 Project Reflection

At the beginning, this project was very challenging to me since I had started coding after only reading the assignment pdf once or twice. I ended up reading the assignment pdf a few other times and restarting my coding process to ensure that I followed good practices and most importantly didn't break encapsulation within my code. The time that I spent on this project was significantly greater than what I have spent on other computer science projects. I feel that this is due to the complexity and large size of this project compared to what I was used to. Personally, I believe that the time period given for this project was very reasonable especially considering the amount of detail covered in the assignment pdf. I can say with unwavering certainty that this project has made me realize how important minor changes in a large program can have on things like encapsulation. I was definitely challenged at various points during this project and I feel that struggling with some parts has allowed me to grow as a programmer.

8 Project Conclusion/Results

The purpose of this project was to implement an interpreter to execute ByteCodes of a language known as 'language X'. I've done my best to add helper functions within my code in order to make it more readable and efficient. Additionally, I put a great deal of thought into the concept of encapsulation and accessibility to ensure that my code is robust and follows good OOP principles. I am proud to say that my project functions satisfies all of the requirements given in the assignment pdf. I've executed both the factorial.x.cod and fib.x.cod files with dumping on various different cases and edge cases to ensure that they are dumping properly and as expected. I have also compared my dumping output for factorial.x.cod with the sample output posted by the Professor on the Slack channel and it completely matches with minor changes(in line with stated assignment pdf requirements). The debugger in the IntelliJ was used extensively throughout my design process to ensure that each part of my project was functioning properly.