

# ***CSC 413 Project Documentation***

***Fall 2018***

## ***The Interpreter Project***

***Student name: Ratna K Lama***

***Student ID: 909-324-382***

***Class.Section: CSC413.01***

***GitHub Repository Link:***

<https://github.com/csc413-01-fa18/csc413-p2-rlama7.git>

## Table of Contents

1	Introduction .....	3
1.1	Project Overview.....	3
1.2	Technical Overview .....	3
1.3	Summary of Work Completed .....	5
2	Development Environment.....	7
3	How to Clone, Import, Configure and Build Project .....	7
4	How to Run your Project.....	12
5	Assumption Made .....	13
6	Implementation Discussion.....	14
6.1	Class Diagram.....	15
7	Project Reflection.....	19
8	Project Conclusion/Results .....	20

# 1 Introduction

The goal of the Interpreter project is to implement interpreter program for the mock language X. The interpreter is responsible for processing bytecodes that are created from the source code files with the extension x. The interpreter and the Virtual Machine will work together to run a program written in the Language X.

## 1.1 Project Overview

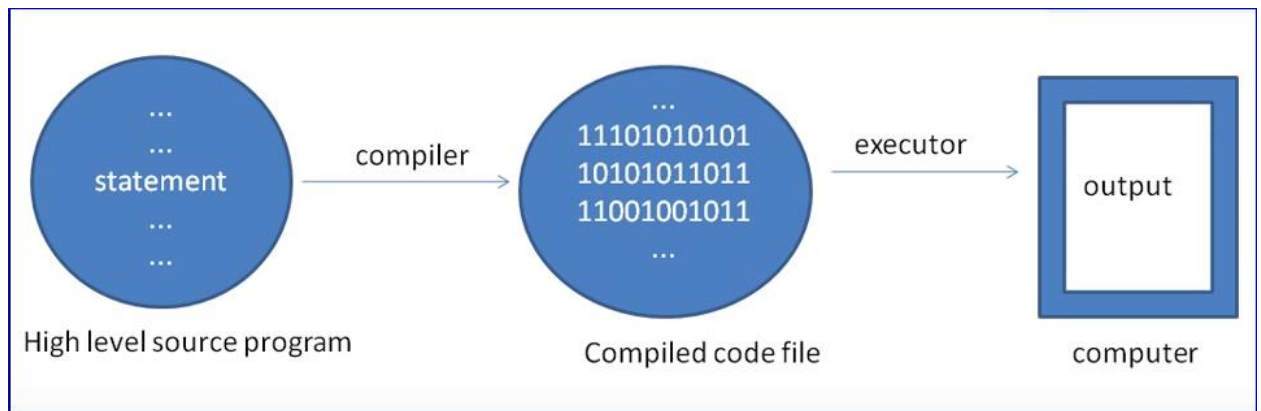
The computer can only understand in the binary form of 0 and 1. However, it is challenging for us human to understand the language formed from strings of 0 and 1. Naturally, when a programmer writes a source program, they are much more likely to be efficient when they program in higher level languages such as C, C++, and Java. These programs are written in the form where it is much easier to read and write to a programmer. Since a computer cannot understand a source program written in the high-level language, a source program must be translated into machine code for execution. The translation can be done using another intermediary programming tool called an interpreter or a compiler.

As an analogy to foreign language, a compiler acts as a translator, and an interpreter acts like an interpreter.

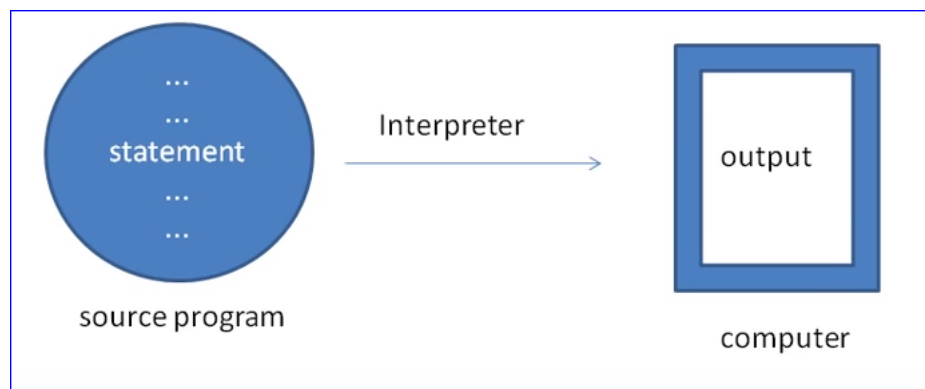
## 1.2 Technical Overview

Java is the first substantial language which is neither indeed interpreted nor compiled; instead, a combination of both are used. This combination of both compiler and interpreter gives Java an advantage of being platform-independence.

The compiler translates the entire source code into a machine-code file or an intermediate code, and that the file is then executed.



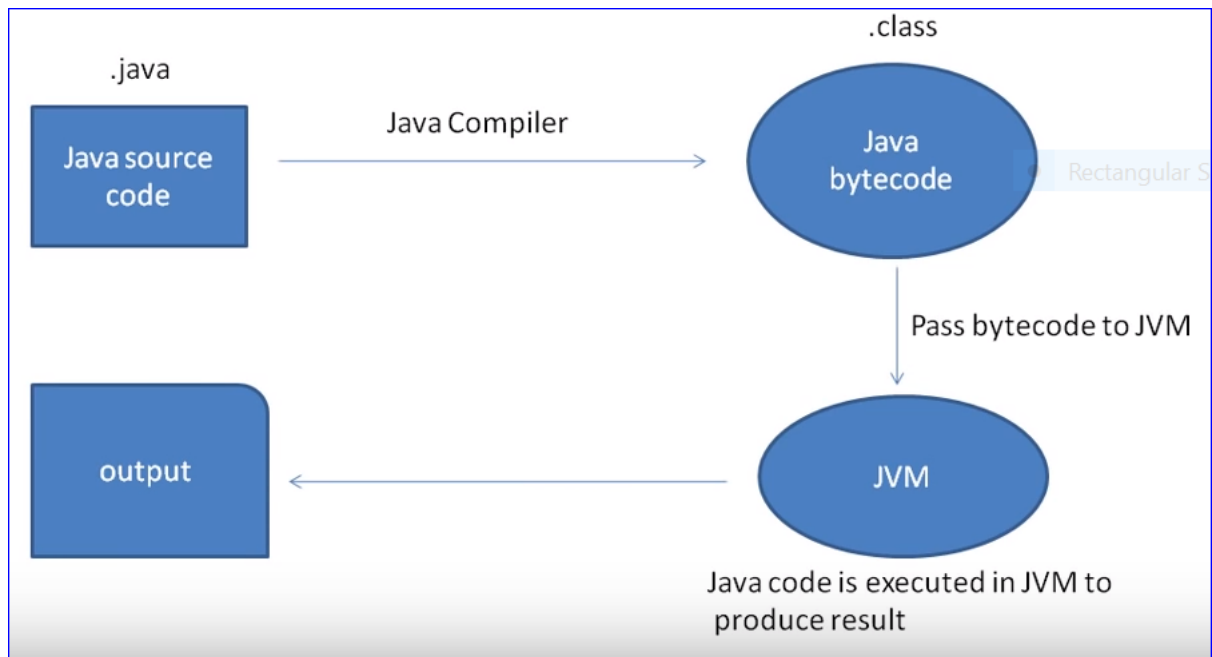
On the other hand, an interpreter reads one statement from the source code, translated it to the machine code or virtual machine code, and then executes it right away.



## How Java Works

First, we can write java source code. It has **.java** extension. Next, we can use the JVM compiler called **javac** to generate a bytecode file (bytecode file uses the extension **.class**). This bytecode file can be used on any platform that has installed Java. However, bytecode file is not an executable file. To execute a bytecode file, we need to invoke interpreter called **java**. After the interpreter is invoked, it will execute the bytecode file and produce output.

Every platform has its own Java interpreter which will automatically address the platform-specific issues.



### 1.3 Summary of Work Completed

First, I implemented an abstract `ByteCode` class and its subclasses. Furthermore, I implemented another abstract `BranchCode` class.

Next, I implemented `ByteCodeLoader` class. This class is responsible for loading bytes from the source code file into a data structure that stores the entire program. The `ArrayList` was used to store bytecodes. The `ArrayList` was contained inside of a `Program` Object. Adding and Getting bytecodes will go through the `Program` class.

Next, I implemented Program class. This class is responsible for storing all the bytecodes read from the source file. ByteCodes are stored in an ArrayList which has a designated type of ByteCode. It is to ensure only ByteCode, and its subclass can only be added to ArrayList.

Next, I implemented RunTimeStack class. This class is responsible for recording and processing the stack of active frames. This class contains two data structures used to help the VirtualMachine Class execute the program. These data structures are maintained private and are as below:

1. Stack Frame Pointer

- a. This stack is used to record the beginning of each activation record (frame) when calling functions.

2. ArrayList<Integer> runStack

- a. This ArrayList is used to represent the runtime stack. It will be an ArrayList because we will need to access ALL locations of the runtime stack.

Finally, I implemented a VirtualMachine class. This class is used for executing the given program. The VirtualMachine is the controller of all the program. All operations need to go through this class.

The Interpreter class and the CodeTable class was provided with implementation. The Interpreter class is the entry point to the Interpreter project and need not any changes. The ByteCodeLoader class uses the CodeTable class. It merely stores a HashMap which allows us to have a mapping between bytecodes as they appear in the source code and their respective classes in the Interpreter project.

## 2 Development Environment

Following development environment made working with the Interpreter project possible:

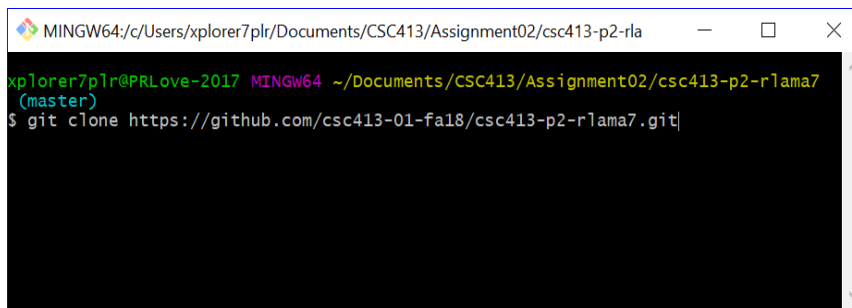
1. IntelliJ IDEA version 2018.2.4 (Ultimate version)
2. Java JDK version 10.0.2
3. Git version 2.14.2 on Windows Operating System (OS) 10
4. UMLet version 14.3, Free UML Tool for Fast UML Diagrams
5. IntelliJ IDE plugin: simpleUML version 0.33,

## 3 How to Clone, Import, Configure and Build Project

1. Steps to clone project (commands are shown in **bold**):

In the git bash clone the Interpreter project from the GitHub repository using the command:

**git clone** <https://github.com/csc413-01-fa18/csc413-p2-rlama7.git>

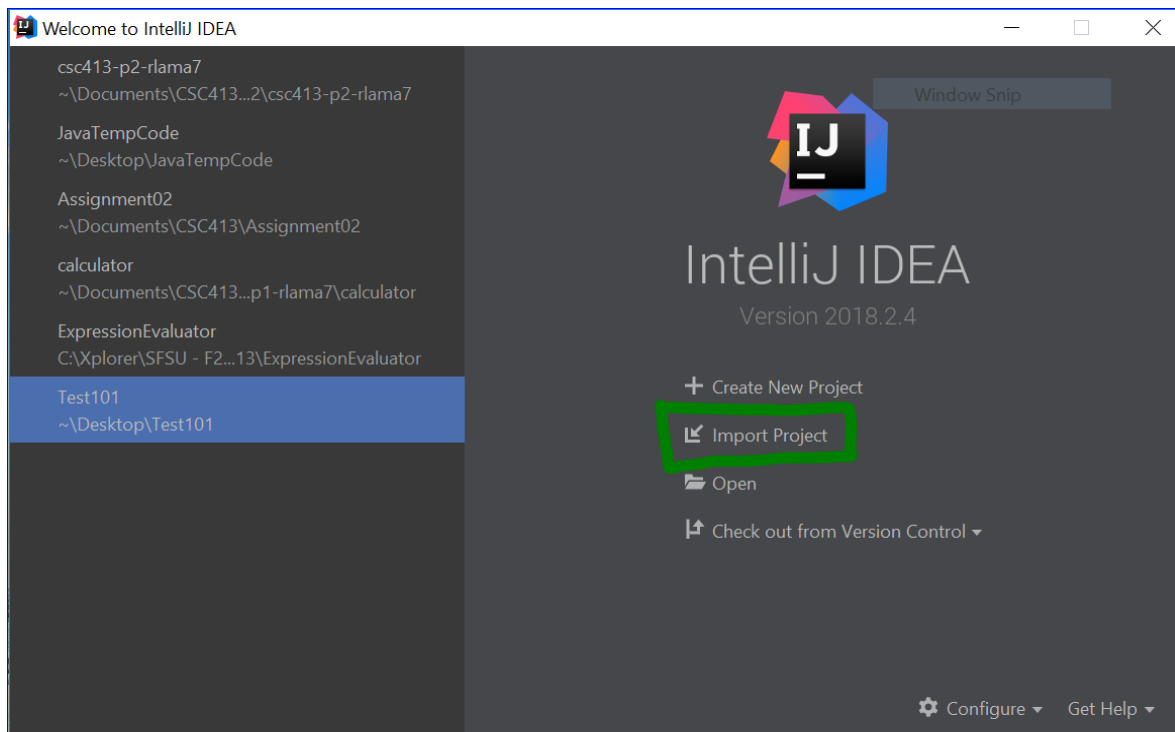


```
MINGW64/c:/Users/xplorer7plr/Documents/CSC413/Assignment02/csc413-p2-rla
xplorer7plr@PRLove-2017 MINGW64 ~/Documents/CSC413/Assignment02/csc413-p2-rlama7
(master)
$ git clone https://github.com/csc413-01-fa18/csc413-p2-rlama7.git
```

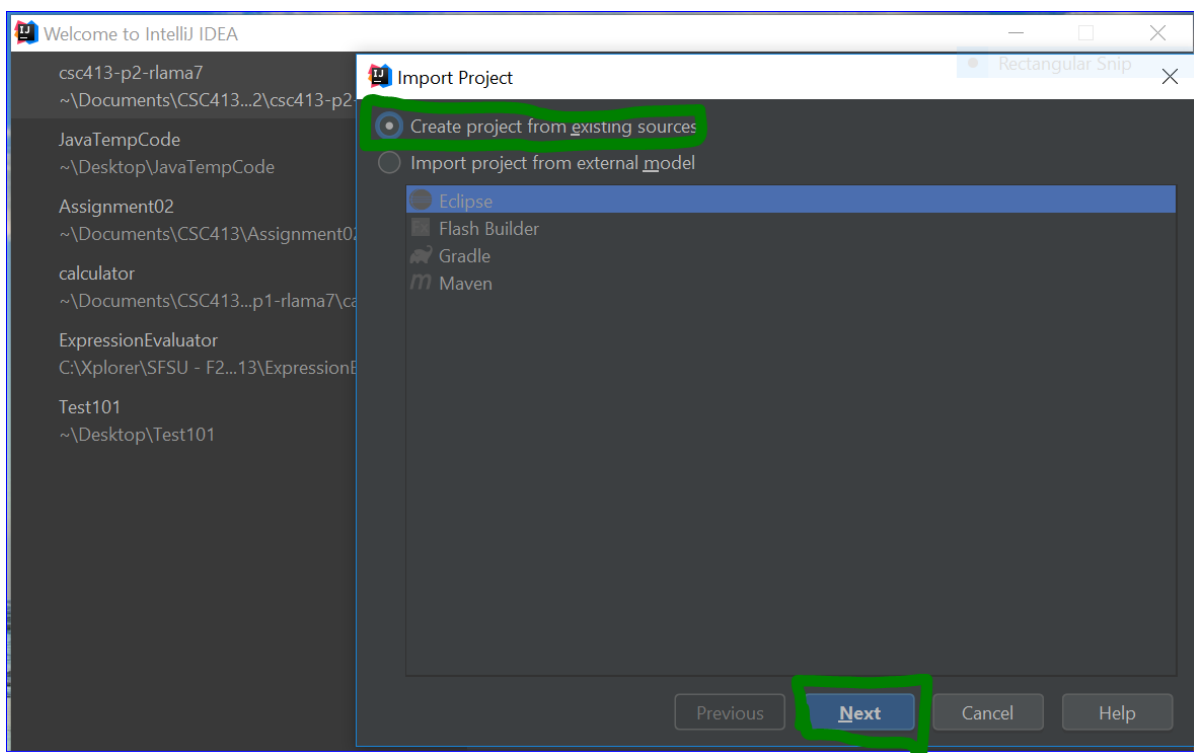
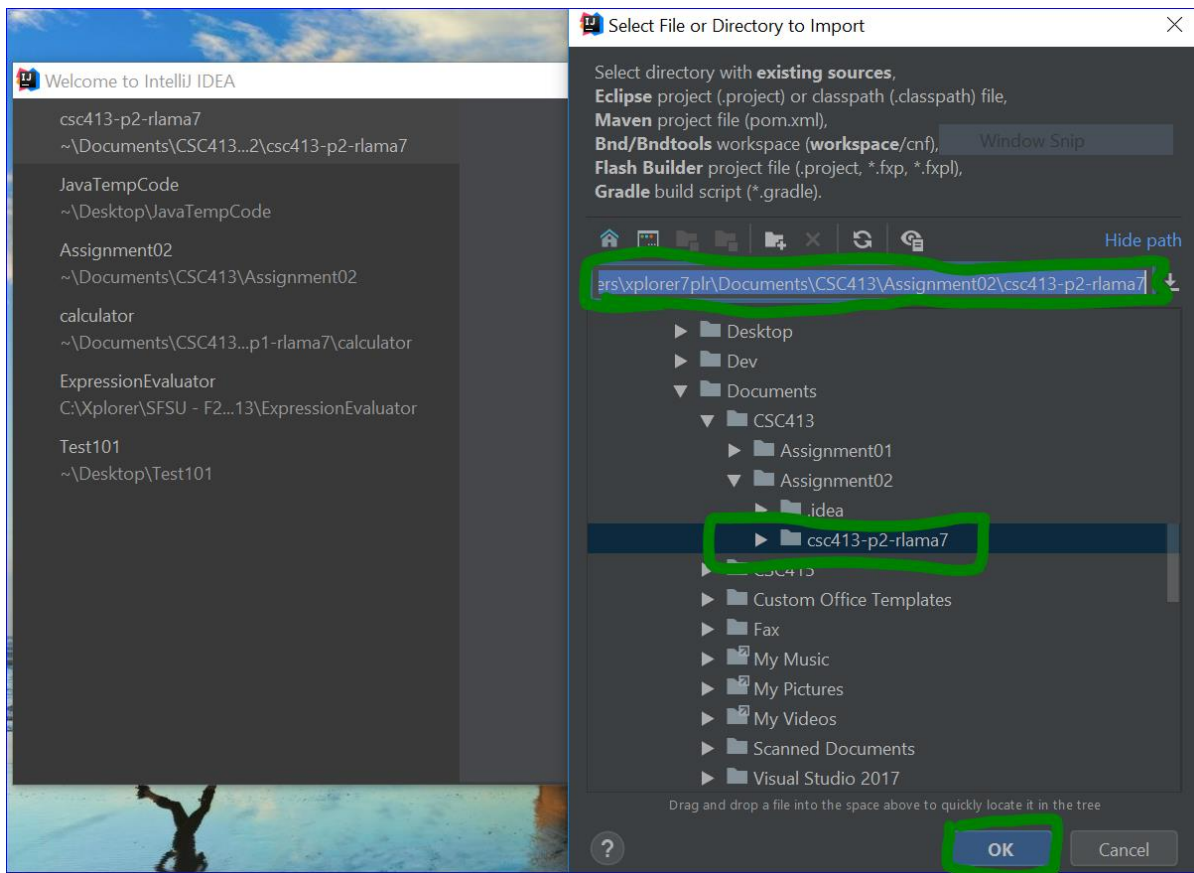
2. Steps to import project (commands are shown in **bold**):

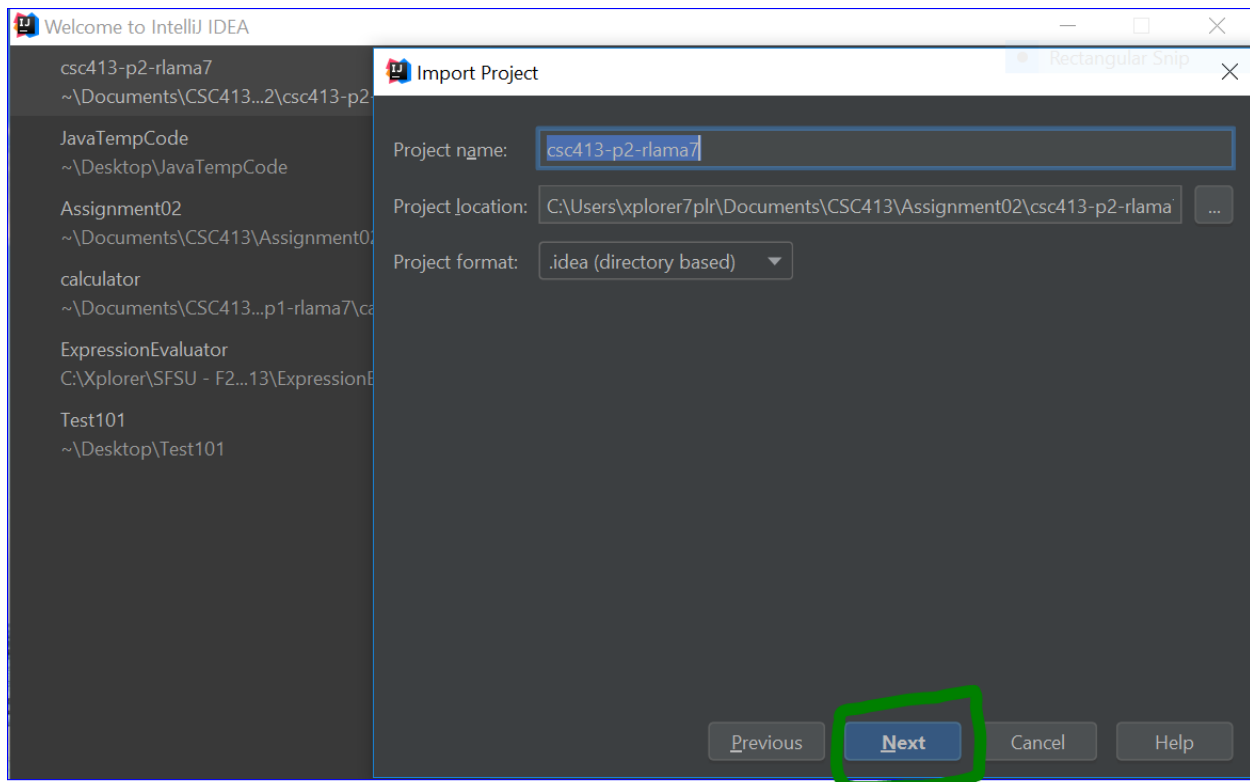
- A. Open IntelliJ and select: **Import Project**
- B. Navigate to the folder where the project folder was cloned. Inside the project folder, navigate to the folder path: CSC413\Assignment02\csc413-p2-rlama7
- C. Next select: **ok**

- D. Next select: **Create from existing resources**. Select **Next** in the IntelliJ IDEA window
- E. Select **Next** again
- F. Select **Finish** to complete the Project Import steps.





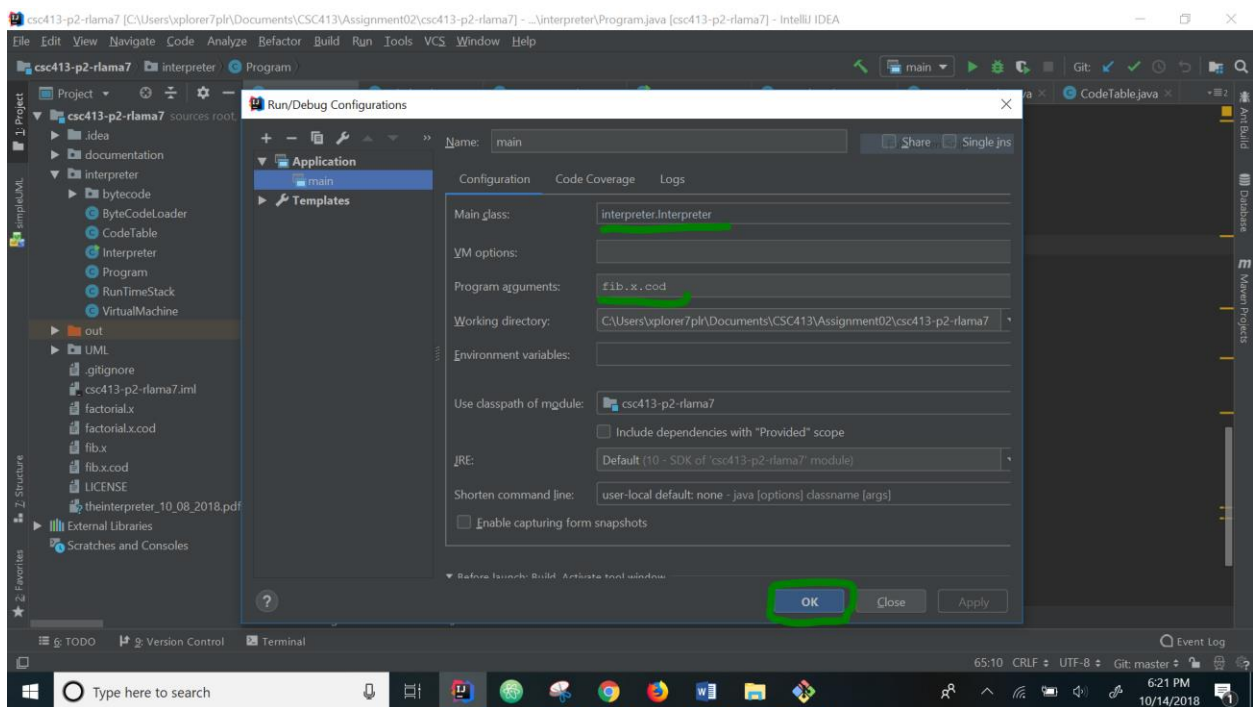
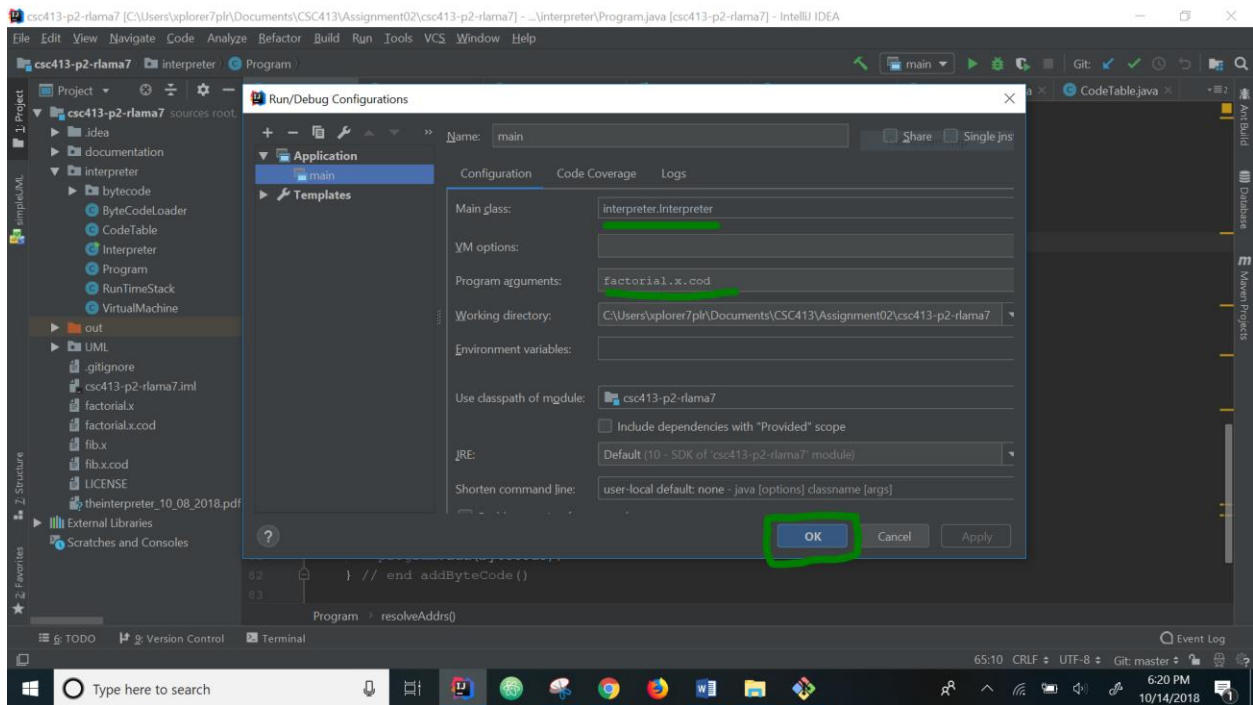




### 3. Steps to configure the program argument:

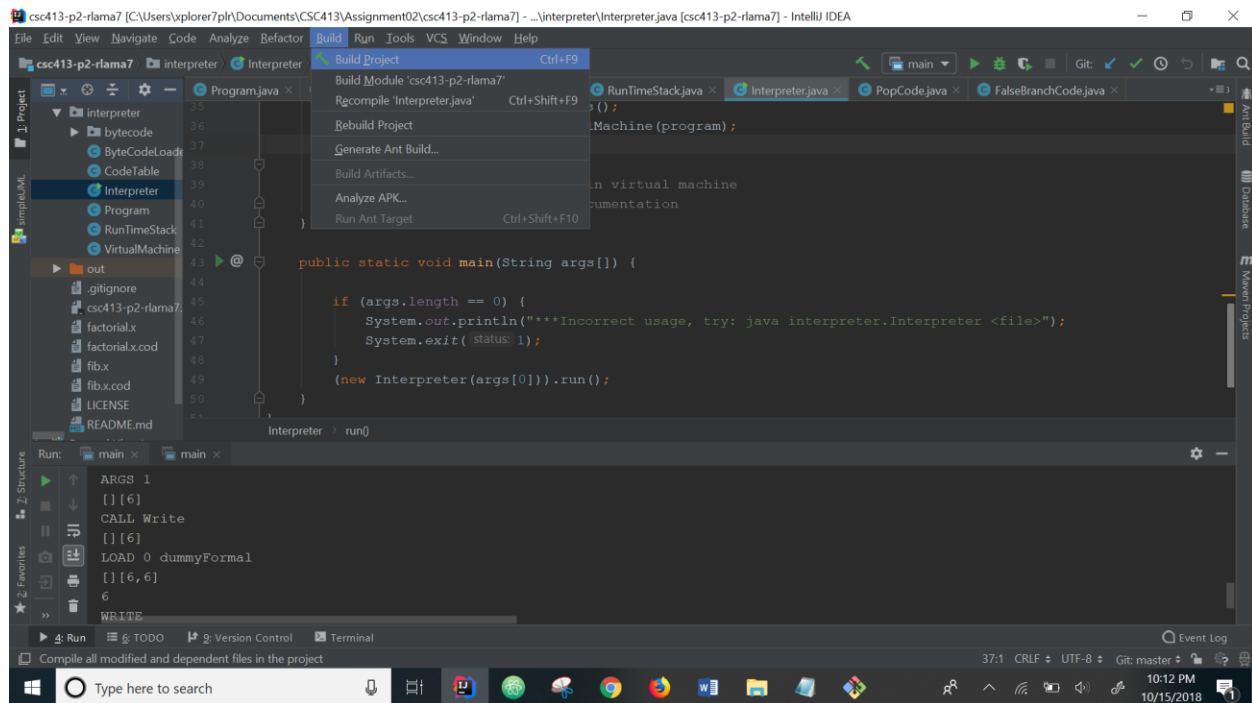
Under configuration tab enter the following information:

- i. Main class: `interpreter.Interpreter`
- ii. Program arguments: `factorial.x.cod`
- iii. IntelliJ IDE should populate the necessary fields.
- iv. Next, select **OK**
- v. Repeat steps (i) to (iv) from above replacing program arguments: `fib.x.cod` to build `fib.x.cod` in step (ii).



#### 4. Steps to build the project:

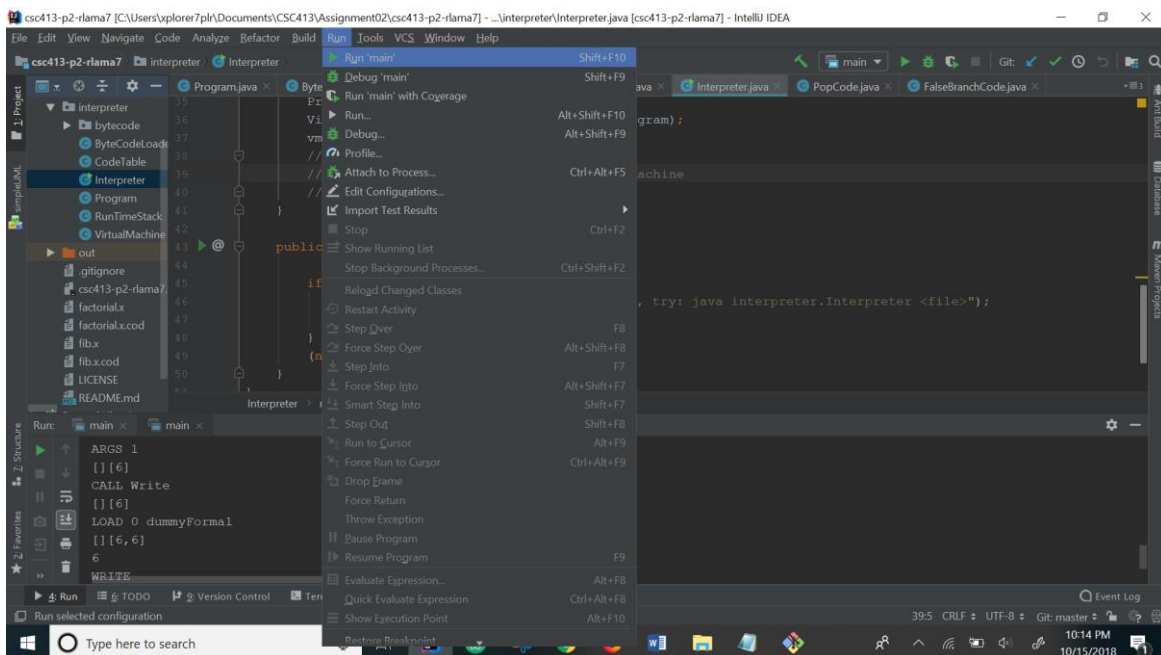
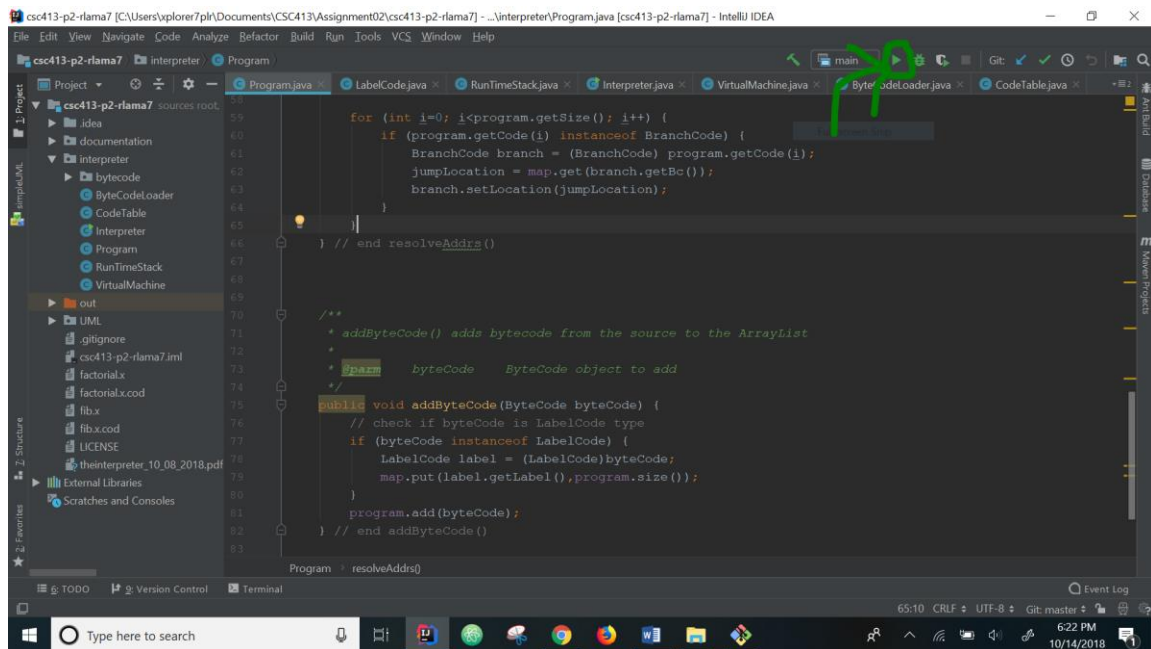
In the IntelliJ IDE under **Build** Menu select: **Build Project** from the drop-down list



## 4 How to Run your Project

At this stage, this project can be Run either by press green play button next main tab or

Pressing **Run 'main'** from the Run tab.



## 5 Assumption Made

I assumed that the two bytecode files: factorial.x.cod and fib.x.cod, which were supplied as an argument to run the program is correct, and no error intended.

## 6 Implementation Discussion

I followed instructors suggested order to implement the program from the documentation.

I was not quite sure why we were required to implement so many different classes. After hours of combing through the program lines, discussing with tutors at ACM tutoring, and many in-class discussions, I noticed a light in the tunnel towards the end of all the program implementations.

All those different classes had a single responsibility which is the first design principle of an Object-Oriented Programming (OOP) design pattern.

Second, all the classes had proper access modifier enforced to ensure they are all open for extension but closed for modification which is the second principle of OOP design pattern.

Third, all the bytecode subclasses inherited from the parent ByteCode class. When a subclass inherits from parent class there exists “IS-A” relationship between a parent and a child class. However, this is not a sufficient condition for inheritance. It is more appropriate to say that one object can be designed to inherit from another if it always enforces “IS-SUBSTITUTABLE-FOR” relationships with the inherited object. It forms the basis for Liskov’s substitution principle (LSV) in the design pattern. The LSV principle of design pattern states that subtypes must be substitutable for their base types.

Fourth, interface segregation principle of the design pattern states that clients should not be forced to depend on methods that they do not use. This principle does not seem to rule in the surface level in this program. However, I could imagine this principle in play in other Java classes that I implemented such as ArrayList and Stack that implements other interfaces.

Finally, dependency inversion principle states that high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions. In our program, the ByteCode class is an abstract class. It does not depend on sub-bytecode-classes.

Generally, software should be designed in such a way that any future changes will not break the entire program. I learned that following those above design pattern principles as a guide to a better design. I hope to master those principles as the software design course advances.

## 6.1 Class Diagram

The Interpreter Class is the entry point to the program.

The ByteCodeLoader Class is responsible for loading bytecodes from the source file into a data structure that stores the entire program.

The ByteCode Class is an abstract class. It abstracts out the implementation details. It consists of two public methods – init and execute which all child ByteCode classes such as HaltCode and PopCode Class must implement.

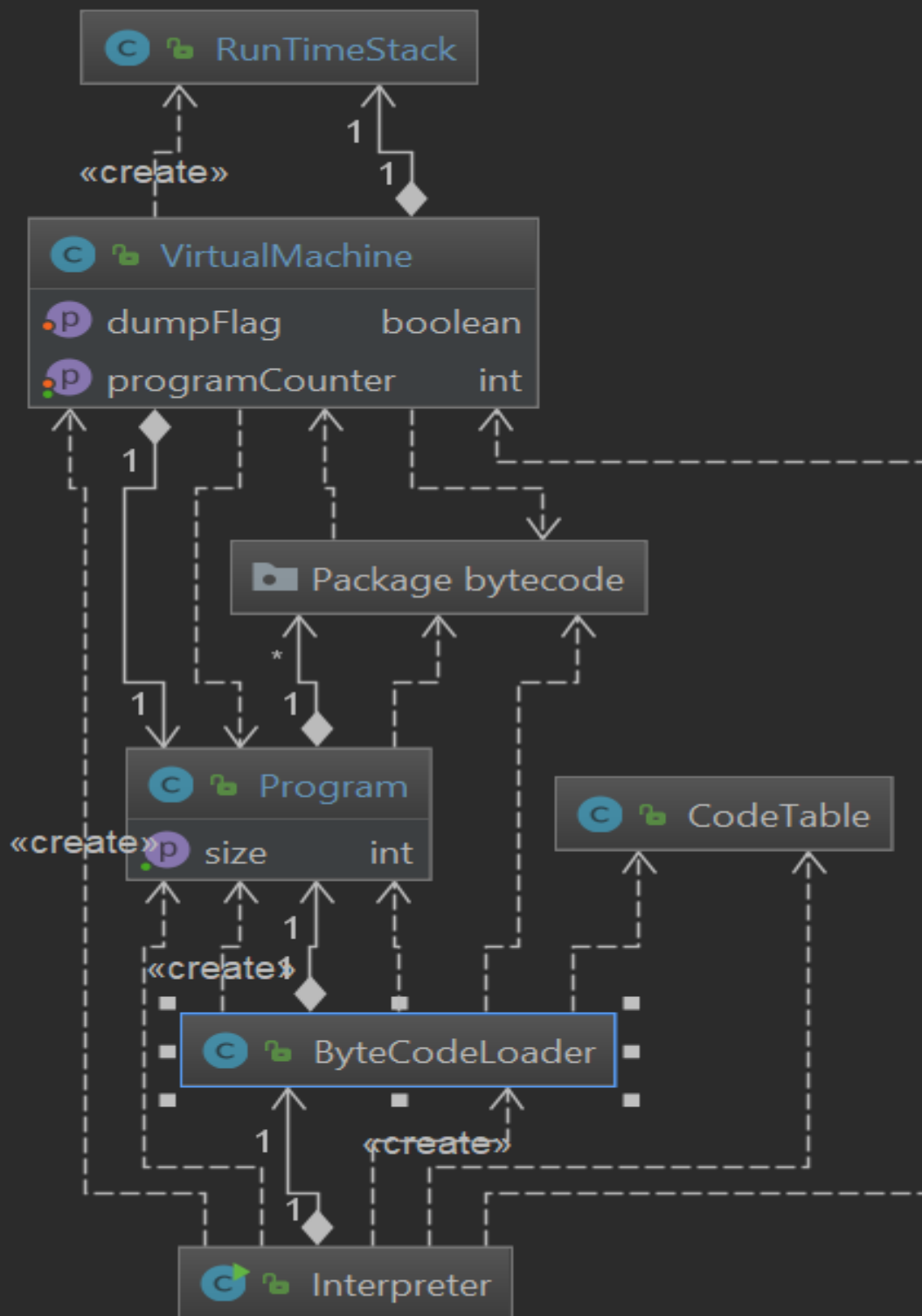
The ByteCodeLoader Class uses the CodeTable Class. It merely stores a HashMap which allows us to have a mapping between bytecodes as they appear in the source code and their respective classes in the Interpreter project.

The Program Class is responsible for storing all the bytecodes read from the source file. We stored bytecodes in an ArrayList which has a designated type of ByteCode to ensure only ByteCodes and its subclass can only be added to the ArrayList.

The RunTimeStack Class is responsible for recording and processing the stack of active frames. There are two data-structures within this class to help the VirtualMachine Class execute the program.

The VirtualMachine Class is used for executing the given program. This class is the controller of this program. All operations need to go through this class.







## 7 Project Reflection

I learned from my first project on expression evaluators that a UML diagram is significant in getting a bird's eye view on the whole project. Therefore, I started out filling UML diagram with free UML diagram tool called UMLet as I read the documentation. To be honest, it was quite a daunting task when trying to decipher twenty-one-page documentation. However, drawing a UML diagram as I progressed on the documentation saved me a lot of hassle. And being a visual person, it was a no-brainer for me that I got the high-level view of the project with the help of the UML diagram.

First, I implemented the ByteCode class which is an abstract class with two abstract methods: init and execute. Next, I implemented all fifteen ByteCode subclasses which inherited from the parent. The subclasses must implement init and execute methods, so I created an empty method, to begin with. Then I started filling those methods as I progressed along with other Classes.

Next, I implemented ByteClassLoader Class. I needed to implement the resolveAddr method in this class. I delegated the resolveAddr method to Program Class since this task was most suited to the class that is responsible for storing all the bytecodes read from the source file.

Next, I implemented Program Class. In the class implementation of the resolveAddr was a significant challenge. The resolveAddr function should go through the program and resolve all addresses. It was crucial because in order for VirtualMachine class to set the Program Counter (pc) all the labels in the bytecodes class need to be first converted to correct addresses. To handle this situation, I created another abstract class BranchCode which inherited from ByteCode Class.

The need for BranchCode abstract class is to handle all the branching scenarios requirements. The FalseBranchCode, the GotoCode, and the CallCode are a child of BranchCode.

Next, I implemented RunTimeStack Class. In this class implementing the dump method was a critical challenge. The dump method is used to dump the current state of the runtime stack class. Moreover, when printing the runtime stack divisions between frames needed to be included. If a frame is empty, that needed to be shown as well. It took me at least two days to work on dump function. Towards the final day, I still had some IndexOutOfBounds error. I visited the instructor's office hour, and he was quick to point out the error. This possibly saved much headache in the aftermath.

Finally, I implemented a VirtualMachine Class. This class is the controller of the program. All operations need to go through this class. One significant method in this class is called an execute method. This method was partially implemented. However, I had to fill in other methods on need basis corresponding to individual ByteCode's subclasses request to the VirtualMachine class.

## 8 Project Conclusion/Results

The interpreter project was undoubtedly a challenging project. Be that as it may, I struggled through many nights and days dreaming and thinking about many possible ways to handle scenarios of dump function, resolveAddr function, and several others. In hindsight, I should have actively participated in class discussion and slack forum asking more questions. I am deeply grateful to the instructor for allowing the entire class to give the final touch to our project with his gracious extension at the final tick of the due date. No matter how challenging the project might be I strive to learn from my mistakes and successes. Moreover, I must

confess I have come a long way in becoming a better programmer in designing better software in this project.