CSC 413 Project Documentation

Summer 2022

Timmy Tram

921102465

CSC413.01

https://github.com/csc413-SFSU-Souza/csc413-p2-TimmyTram

Table of Contents

1 Introduction

1.1 Project Overview

The purpose of this project was to create a special program called an interpreter which translates a programming language into operations which the computer can evaluate. In the case of our project, we are interpreting a mock programming language called X.

1.2 Technical Overview

The project's focus was to implement an interpreter that can read and run programs written in a mock programming language X's byte codes created from the source code files. To achieve this, I had to implement the bytecode loader where its responsibility is to read the given .cod file and convert it into a Program object where each operation is then converted into some form of a bytecode class. The virtual machine class then receives the given program from the bytecode loader and begins executing each operation and acts as the middleman between the runtime stack and each bytecode operation to promote encapsulation and low coupling. The runtime stack performs similar operations to a normal stack except we have frames or partitions to represent multiple function calls. Lastly, the ByteCode abstraction and each of its children implement some form of initialization and execute function to perform their respective jobs like WriteCode printing to the console.

1.3 Summary of Work Completed

In this project I was able to complete:

- The Virtual Machine's functionality of interacting as the middleman between the runtime stack / return address stack and bytecodes. I was also able to implement dumping within the executeProgram() function.

- The Runtime Stack's functionality of managing frame boundaries, peek, pop and pushing as well as implementing the dump functionality of the runtime stack with the proper frame boundaries being displayed.

- The ByteCodeLoader's functionality of reading the given .cod file and converting each line into some bytecode that our Program class can read and pass to the Virtual Machine.

- The Program class's functionality of reading all ByteCodes that require addresses and what location in the .cod file to go to. This means LabelCode, CallCode, GotoCode, and FalseBranchCode's addresses work as intended.

- Additionally, I created the Dumpable marker interface to distinguish which bytecodes are allowed to be dumped / printed onto the console.

- Also created the BranchCode abstract class which extends the ByteCode abstraction to encapsulate and reuse functions that are shared between the bytecodes that need their addresses resolved.

- Lastly, I designed a simple abstraction of ByteCode since it is more of an idea that is not meant to be instantiated and more serves as a base for all other bytecodes such as:

1. ArgsCode
2. BopCode
3. DumpCode

4. HaltCode
5. LabelCode
6. LitCode
7. LoadCode
8. PopCode
9. ReadCode
10. ReturnCode
11. StoreCode
12. WriteCode
13. BranchCode ← Abstract class that extends ByteCode
    a. CallCode
    b. FalseBranchCode
    c. GotoCode

2 Development Environment

a. Version of Java used: openjdk-16 or Java Version 16.0.2
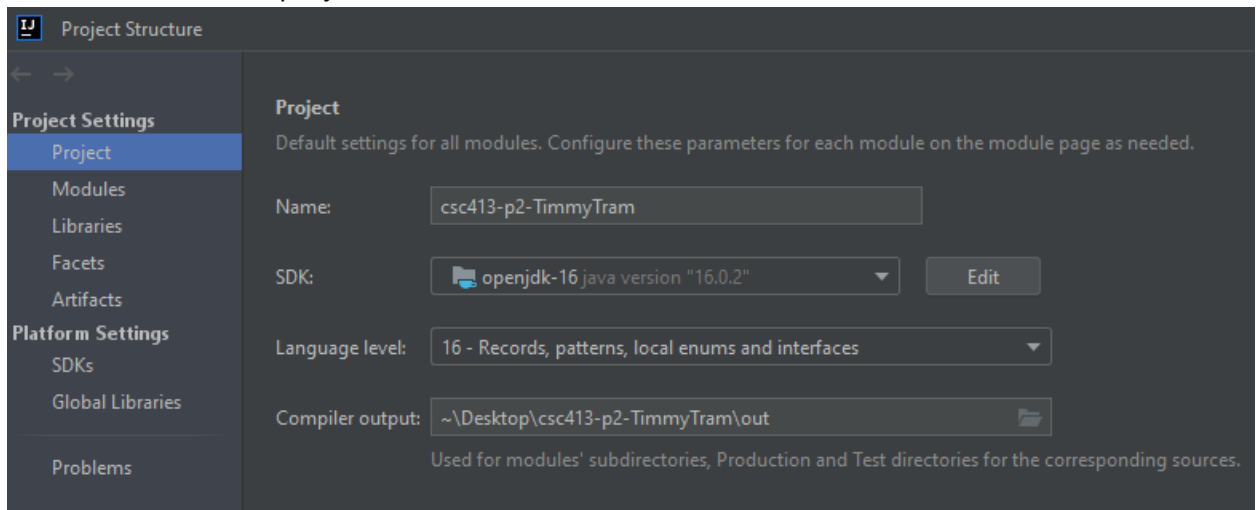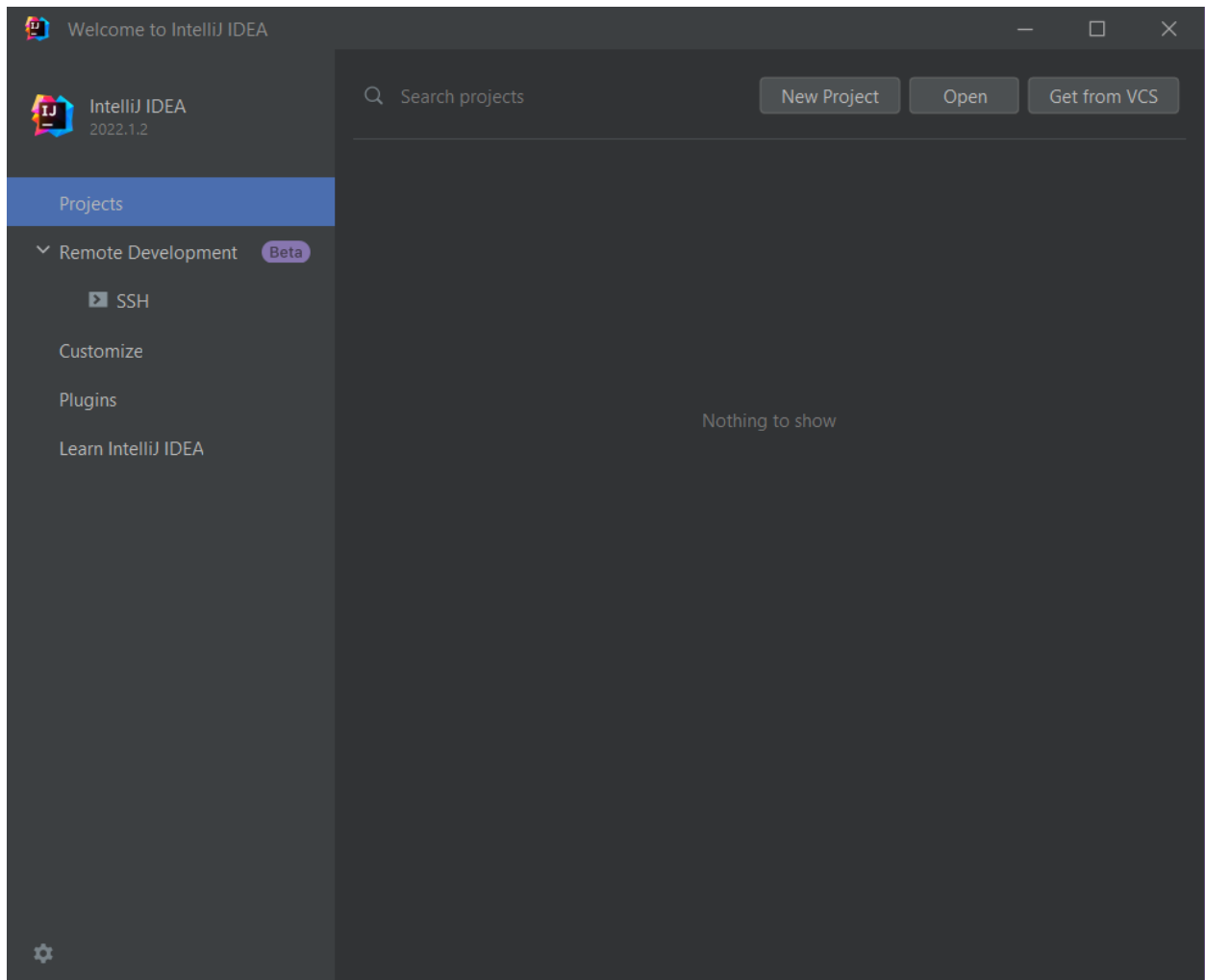

b. IDE used: Intellij IDEA Ultimate Edition 2022.1.2
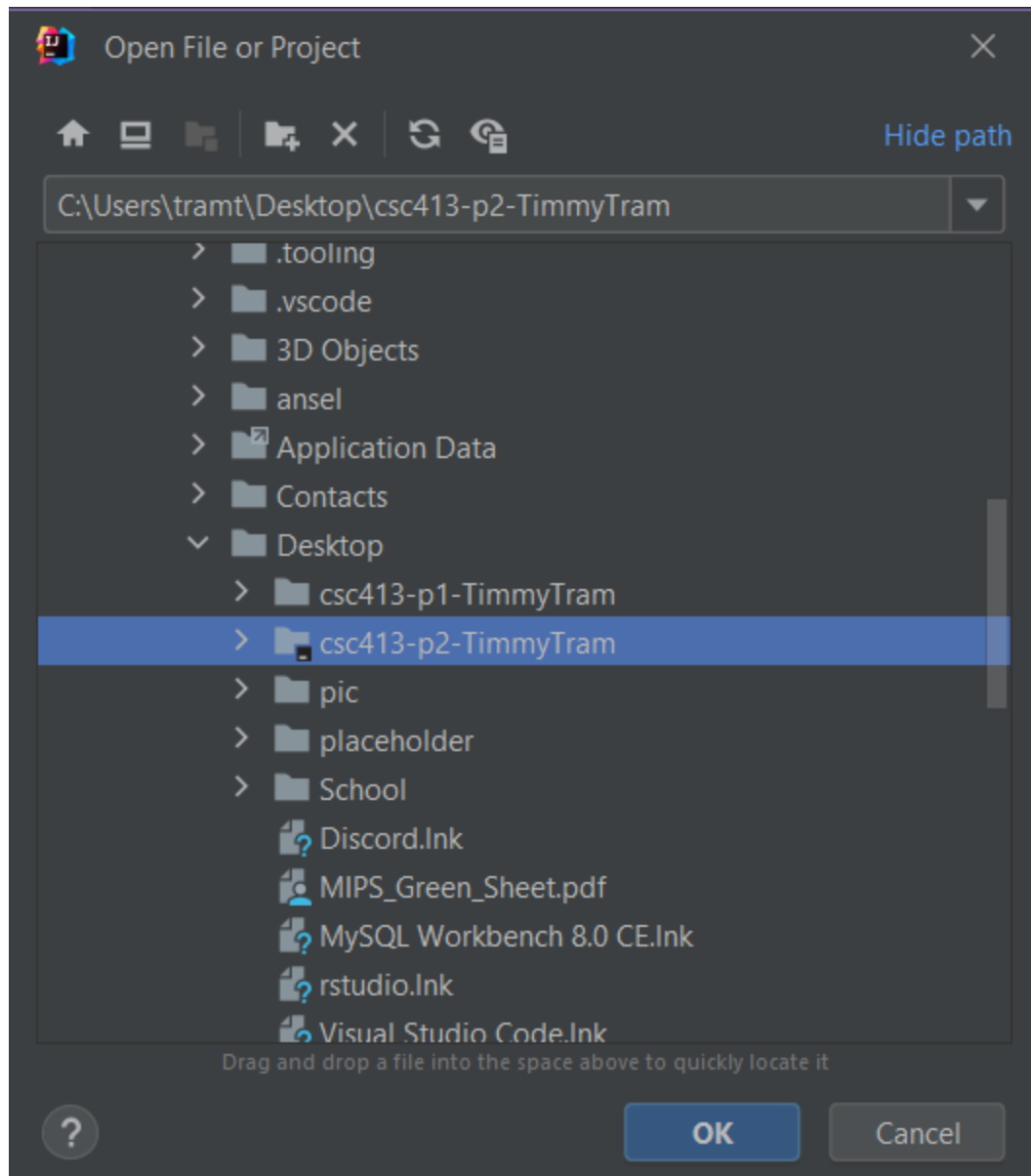
3 How to Build/Import your Project

Open a terminal of your choice and type: git clone https://github.com/csc413-SFSU-Souza/csc413-p2-TimmyTram.git
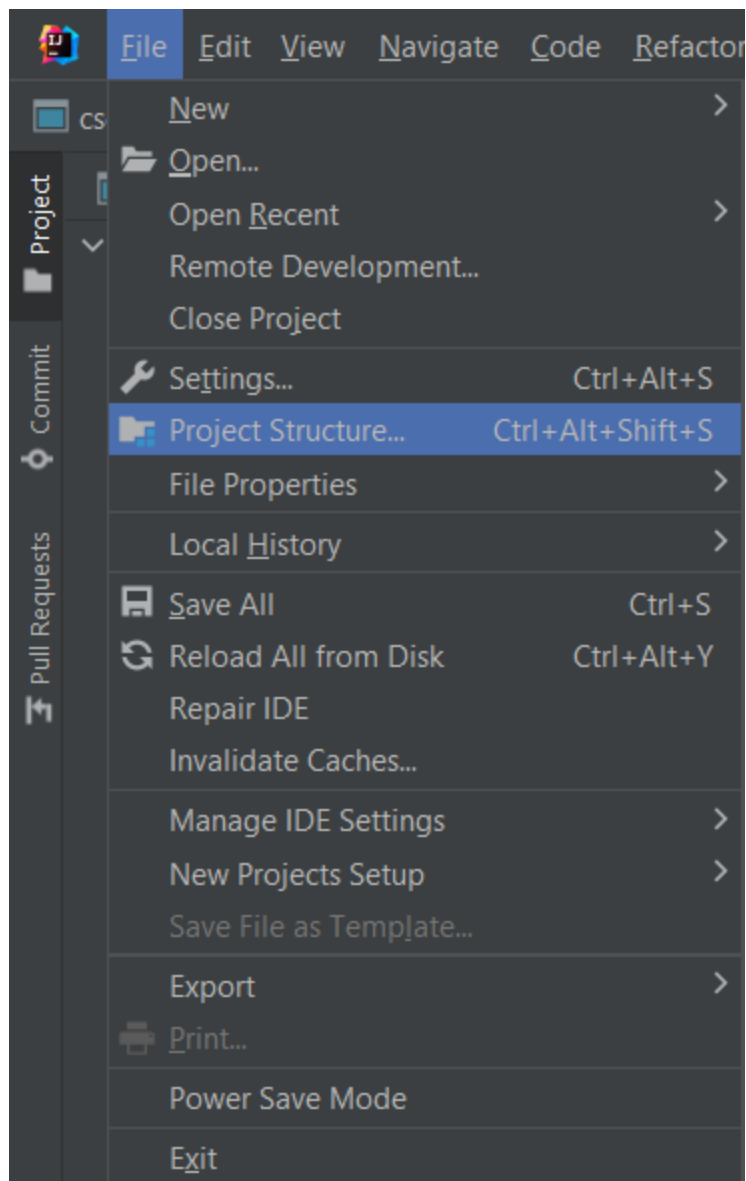

Select Open on Intellij IDEA:

Select csc413-p2-TimmyTram as the source folder and press ok

Then in the top left click File and then click Project Structure

Then in Project Structure make sure the SDK is some version of Java 16 and set the language level to SDK default (16 – Records, patterns, local enums and interfaces) and then hit apply and ok.
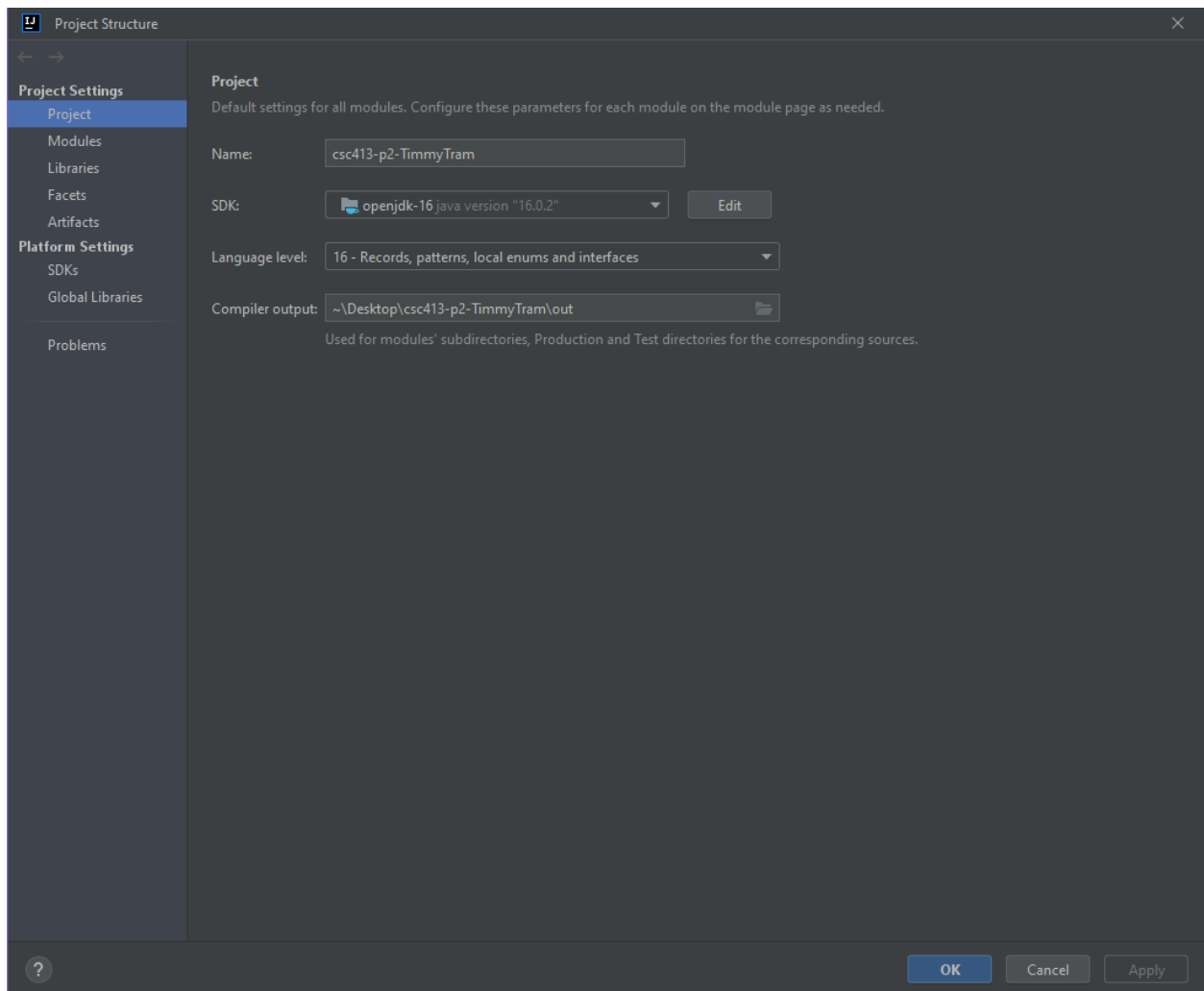
4 How to Run your Project

After successfully importing the project and configuring the SDK you must find Interpreter.java located in the interpreter package under the source of the project. Then right click the Interpreter.java file and hit run.
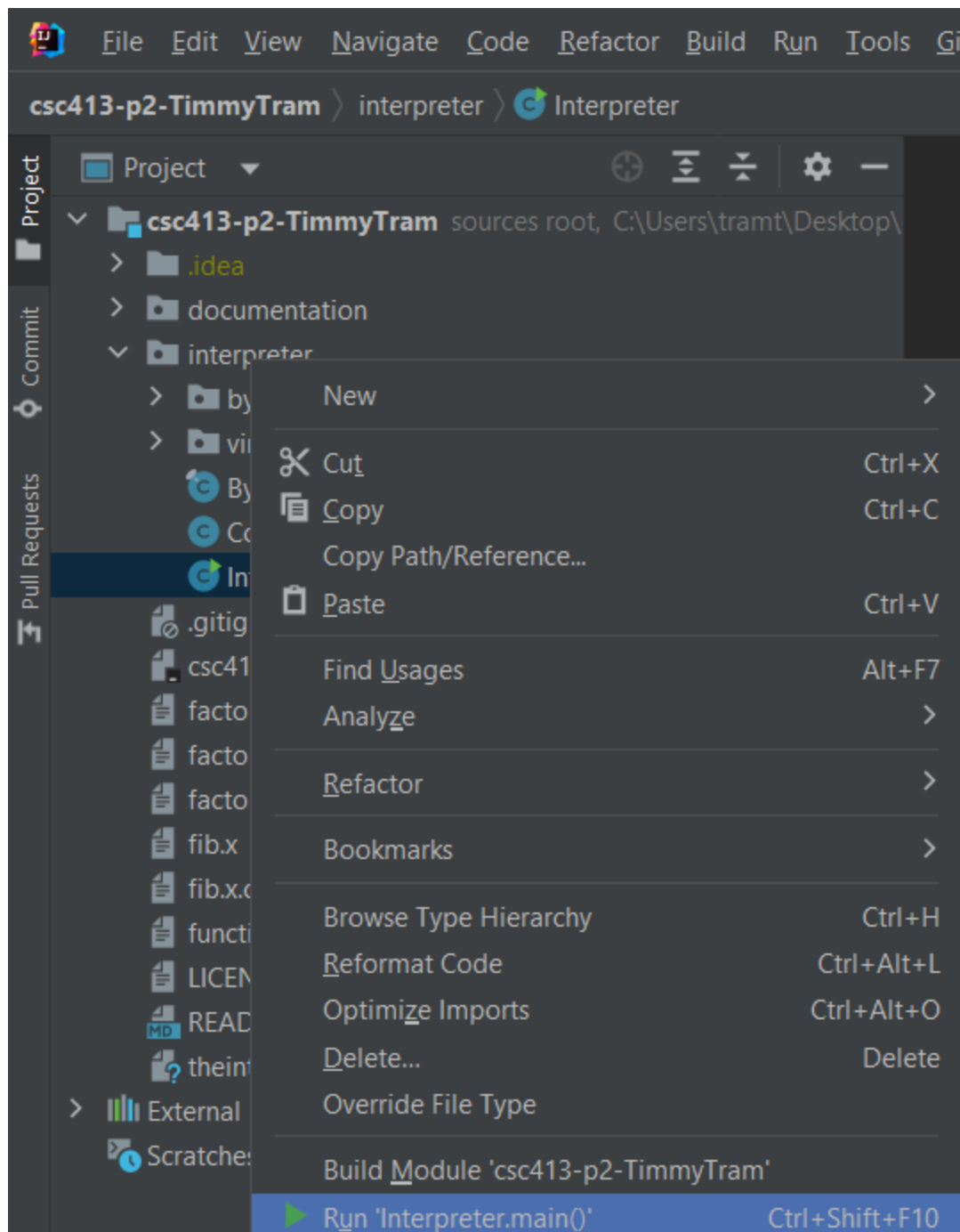
You will then see this output with exit code 1 meaning the program works, but we need to add arguments.

On the top bar you should see this icon and you want to click on it and then click Edit Configurations.
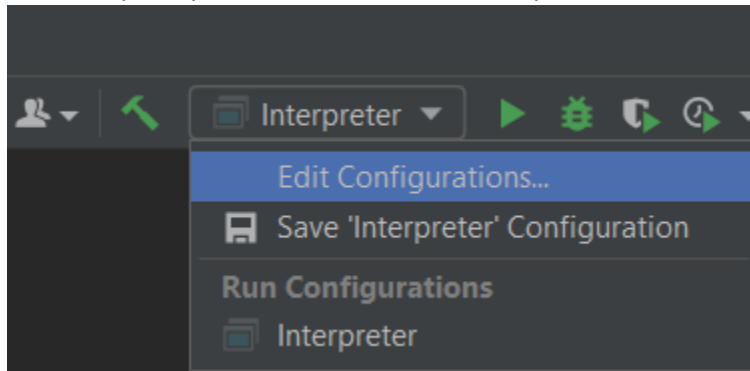
You will then be presented with this window and where it says Program Arguments you simply add the .cod files you want to be interpreted.

The following files that can be interpreted and passed as Program Arguments are:

1. factorial.dump.cod
2. factorial.x.cod
3. fib.x.cod
4. fib.dump.cod ← Custom file I created to make sure dumping works for fib
5. functionArgsTest.cod
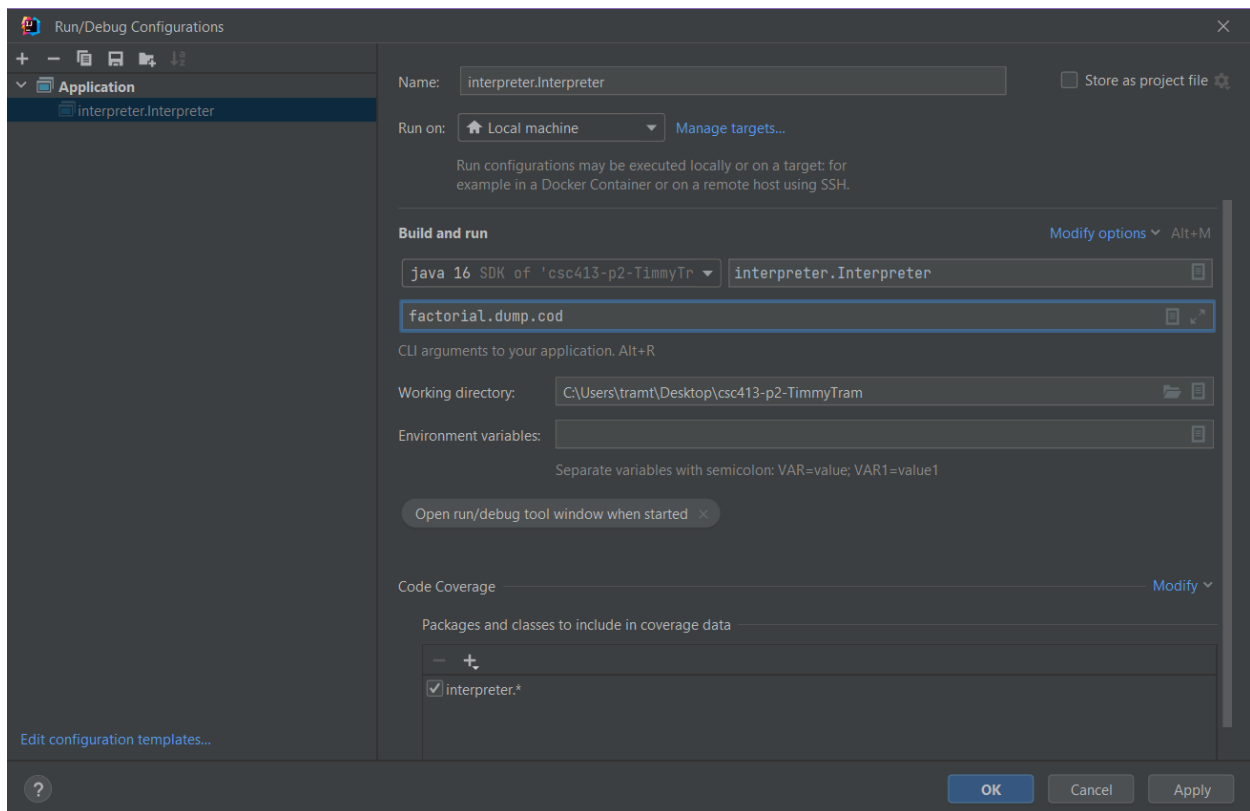
Then simply run the interpreter program again as shown:

And now the program will run the given .cod file.

```
C:\Users\tramt\.jdks\openjdk-16.0
[]
GOTO start<<1>>
[]
GOTO continue<<3>>
[]
ARGS 0
[] []
CALL Read        Read()
[] []
Please enter an integer : |
```

5   Assumption Made

I assumed the given cod files are valid program written in mock language X. (e.g: no extra spaces or words stuck together or typos in bytecode names)

I assumed the user input are integers which can technically be negative although in the case of Fibonacci or factorial it will simply return the same thing as the base case. Al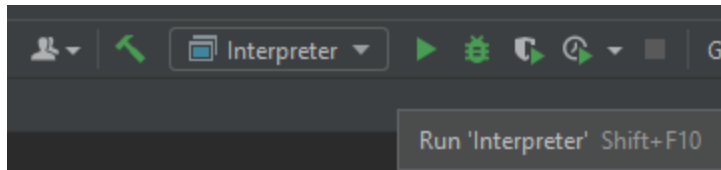so, I assume we are entering reasonably sized integers that will not cause an overflow which can easily happen with factorials.

6   Implementation Discussion

- **Virtual Machine:**
  - The Virtual Machine's sole responsibility was to act as the middleman between the bytecodes and runtime stack. Meaning if a byteCode needs access to runtime stack it should not be able to directly manipulate or use methods given by the runtime stack. The virtual machine will handle the manipulation of the runtime stack and give each bytecode access to functions implemented by the virtual machine class that act as a wrapper for the runtime stack functions. We do this to promote encapsulation which is hiding details and providing methods.
- **RunTimeStack:**
  - The runtime stack implements the basic operations of stack such as peek, pop, and push, but also several new methods to deal with dumping the runtime stack to the

console and methods that deal with creating frames or popping them. Most of the frame related functions use some form of math by using an offset parameter and the size of the runtime stack or the top value of the framePointer to calculate what to do.
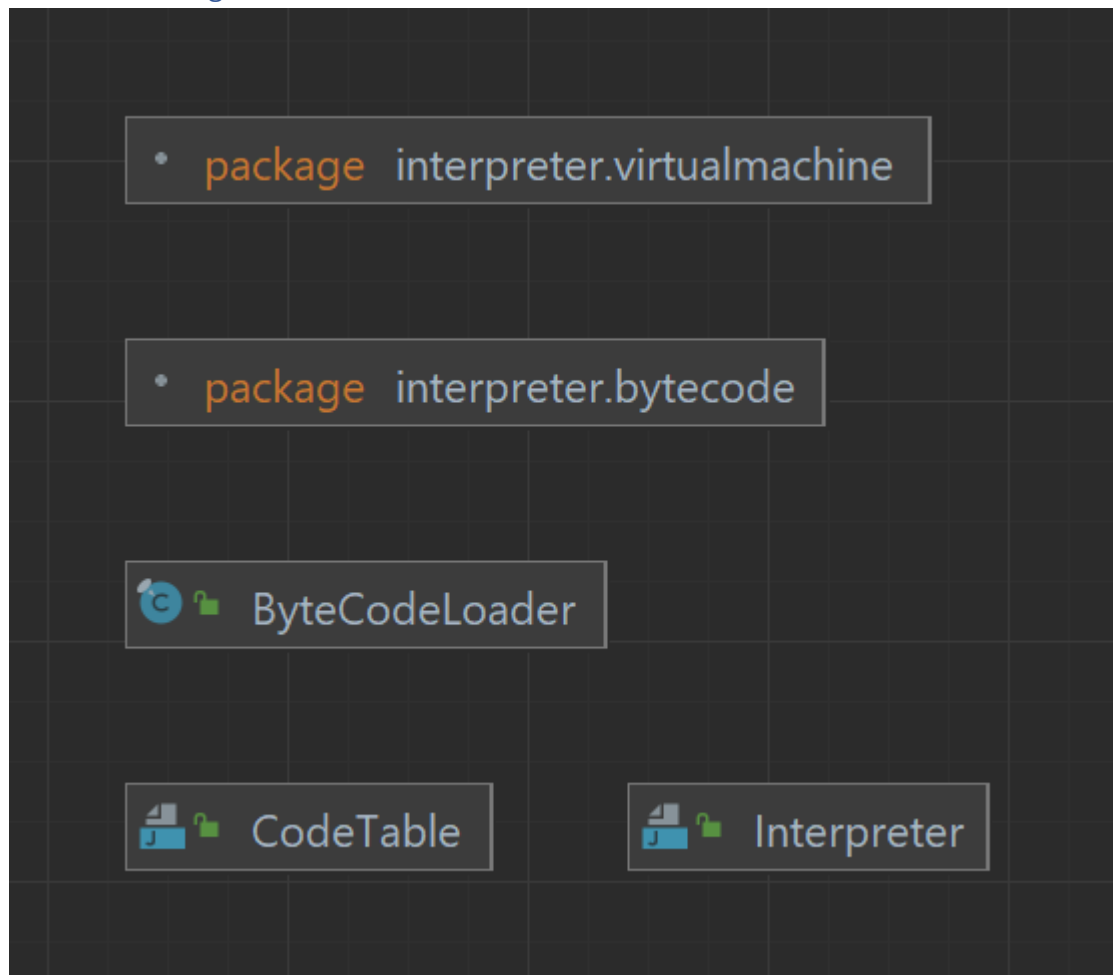
- o I also created a main method within this class as a form of testing functionality of the runtime stack class. It should provide enough information to show that the dumping of the runtime stack is working as intended.

- **ByteCodeLoader:**
  - o We were given implementation of this Class's function loadCodes() via video lecture as well as in-person lecture. But essentially, we use a loop to check if there are still lines left to be read and create the associated objects for each line and add it the program object.

- **Program:**
  - o The main function of Program I needed to implement was resolveAddress(). This function does two for loops through the program's array List of bytecodes. In the first loop we are simply looking for label codes and putting it into a HashMap of string and integers. In the second loop we are searching for any bytecodes that extends the BranchCode abstract class, to figure out where each of these bytecodes specific locations are. Once, we have obtained the addresses we are looking for we set each of the BranchCode's intended jump addresses.
  - o Using the abstract class BranchCode is better if we intend to add more bytecodes that need to have addresses resolved because then we do not have to add an additional if statements to resolveAddress() in order for the program to run correctly. Instead, if we know that a bytecode needs their address to be resolved we should simply have that bytecode extend BranchCode instead of the ByteCode abstract class

- **Dumpable:**
  - o The Dumpable interface is supposed to be used as a marker interface much like Cloneable or Serializable. This interface marks if a certain bytecode is allowed to be dumped by the virtual machine if and only if the bytecode implements the Dumpable interface.
  - o Certain bytecodes, such as DumpCode are not allowed or permitted to be printed to the console, so initially I just checked if a bytecode were a DumpCode and if it were, it would not be printed. This was an inefficient solution, because hypothetically, if I created another bytecode that also cannot be dumped, I would have to add another conditional. This can in turn happen more than just once and I may have 100 conditionals in the virtual machine which I do not want the user to edit.
  - o To combat this problem, I created the Dumpable interface and simply just checked if the bytecode I am currently checking implements Dumpable and if it does then we are allowed to print it.

- **ByteCode:**
  - o The ByteCode is an abstract class because it is more of an idea that is not meant to be instantiated. It contains two abstract methods which are init and execute.
    - ▪ Init takes in a list of strings as a parameter which will be used by each child of bytecode to initialize itself with specific arguments if needed.

- Execute takes in a virtual machine object as a parameter which will be used as our middleman to make requests to the runtime stack to promote encapsulation.
  - **ArgsCode:**
    - Takes in one argument which is how many values will be on a new frame on the runtime stack.
    - Implements Dumpable marker interface.
  - **BopCode:**
    - Takes in one argument which is an operator sign and does logical statement on two operands.
      - The execute function uses and extended switch case that appears in Java 14 and above versions.
      - At first, I implemented this uses just if else statements, but when it came to logical operators '==' it became a nested if statement and became difficult to read at just one glance.
      - Thus, I opted for an extended switch with ternary operators so we can do each operator in just a single line making it easier to see what each operator does in just a single glance or need to scroll.
    - Implements Dumpable marker interface.
  - **CallCode:**
    - Takes in one argument which is a location it must jump to.
    - The toString() method is used for dumping and has a null check to make sure to return the full dump if a label is associated with the CallCode.
      - toString() has a special value it uses called arguments which we receive from the virtual machine using getArgumentsFromFrame() which is a wrapper for the runtime stack. The purpose of this function is to return the values in the runtime stack in a certain frame used for the CallCode.
    - The execute function uses a little bit of regex to make sure that when we get values from the runtime stack via the virtual machine, we do not end up with boundary errors. Otherwise, we simply push our current program counter to the return address stack and jump to somewhere else in the program.
    - Contains a main method to test the dumping format.
    - Implements Dumpable marker interface.
    - Extends the BranchCode abstract class to make use of several getter and setters related to addresses.
      - This cleans up the resolveAddress() code and now we simply just check if a bytecode extends the BranchCode class, we know that it is meant to be jumped towards a different address or location.
  - **DumpCode:**
    - Requests the virtual machine to turn on or off dumping.
    - Does not implement Dumpable marker interface to avoid printing to console.
  - **FalseBranchCode:**
    - Like callCode it takes one argument which is where it is supposed to jump.
    - Implements Dumpable marker interface.

- Extends the BranchCode abstract class to make use of several getter and setters related to addresses.
  - This cleans up the resolveAddress() code and now we simply just check if a byteCode extends the BranchCode class, we know that it is meant to be jumped towards a different address or location.
- **GotoCode:**
  - Like callCode and FalseBranchCode it takes one argument which is where it is supposed to jump.
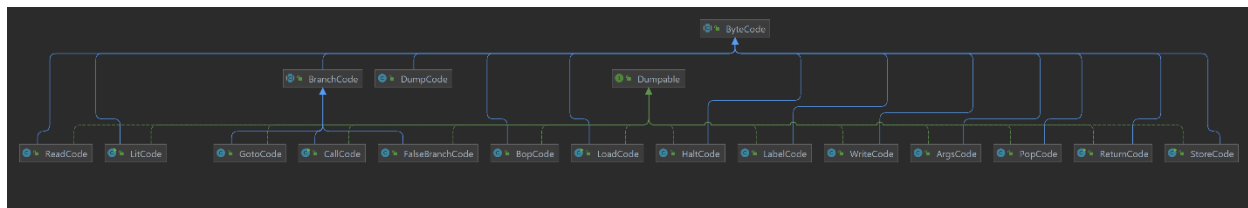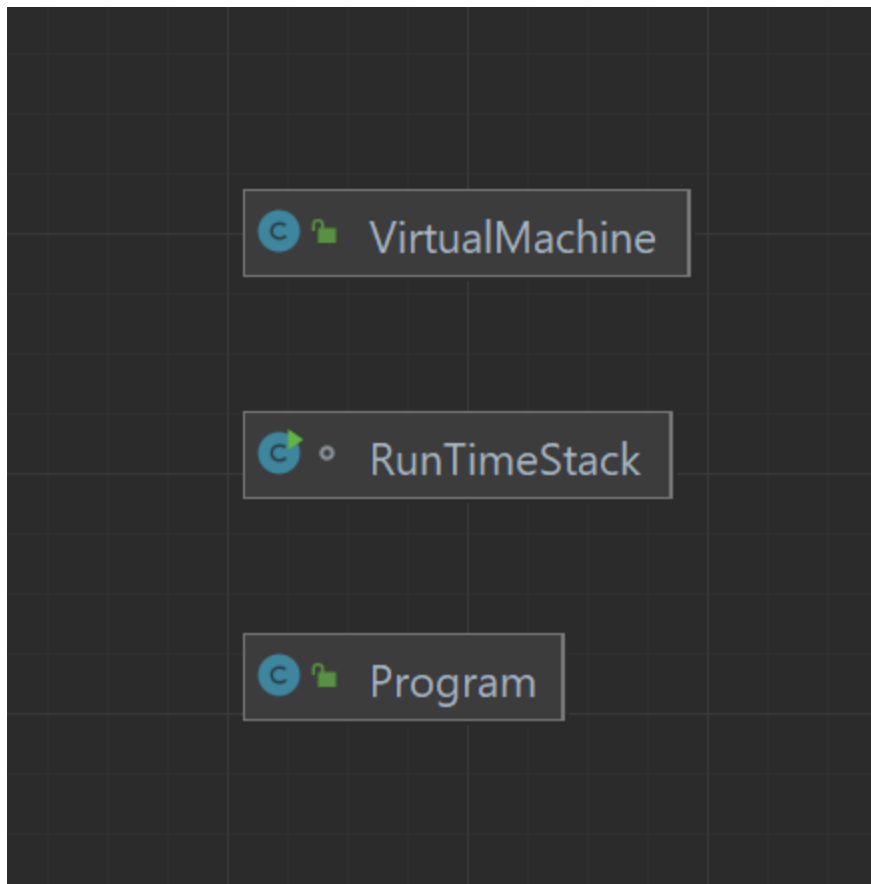 - Implements Dumpable marker interface.
  - Extends the BranchCode abstract class to make use of several getter and setters related to addresses.
    - This cleans up the resolveAddress() code and now we simply just check if a byteCode extends the BranchCode class, we know that it is meant to be jumped towards a different address or location.
- **HaltCode:**
  - Requests the virtual machine to stop executing by changing the value of the Boolean isRunning to false, thereby stopping the while loop in executeProgram() meaning we effectively stopped the entire program without calling System.exit().
  - Implements Dumpable marker interface.
- **LabelCode:**
  - Simply just stores an argument as a label and used in resolveAddress() in the Program class.
  - Implements Dumpable marker interface.
- **LitCode:**
  - Can take 1 or 2 arguments so we add an if statement to avoid index out of bounds errors.
  - Simply push the value we get as the first argument to the runtime stack via request to the virtual machine.
  - toString() has a special dumping format and we use an if statement to do a null check to see if the 2nd argument exists and if it does then build the full dump format.
  - Contains a main method to test the dumping format.
  - Implements Dumpable marker interface.
- **LoadCode:**
  - Can take 1 or 2 arguments so we add an if statement to avoid index out of bounds errors.
  - Simply load the value we get as the first argument to the runtime stack via request to the virtual machine.
  - toString() has a special dumping format and we use an if statement to do a null check to see if the 2nd argument exists and if it does then build the full dump format.
  - Contains a main method to test the dumping format.
  - Implements Dumpable marker interface.

- **PopCode:**
  - Takes in one argument which represents how many times we can pop.
  - Execute function uses Math.min to make sure we do not cross frame boundaries.
    - We do this by getting the number of values in the current frame by making a request via the virtual machine to the runtime stack. Then we use a for loop that loops until we removed a valid number of items on the runtime stack.
  - Implements Dumpable marker interface.
- **ReadCode:**
  - Execute function creates a scanner object and we use a do while loop to keep prompting the user to enter a valid integer by using regex. We then push the user's valid integer onto the runtime stack via request to the virtual machine and close the scanner object.
  - Implements Dumpable marker interface.
- **ReturnCode:**
  - Can take 0 or 1 argument so we add an if statement to avoid index out of bounds errors.
  - The execute function makes several requests to the virtual machine to get the return location as well as get the value from the current frame to hold onto as we pop off that frame and eventually push back onto the frame below where it came from. Then we simply jump back to the return location we requested.
  - toString() has a special dumping format and we use an if statement to do a null check to see if the argument exists and if it does then build the full dump format.
  - Contains a main method to test the dumping format.
  - Implements Dumpable marker interface.
- **StoreCode:**
  - Can take 1 or 2 arguments so we add an if statement to avoid index out of bounds errors.
  - Execute just needs to store some offset and we need to get the current value off the runtime stack via request to the virtual machine
  - toString() has a special dumping format and we use an if statement to do a null check to see if the 2nd argument exists and if it does then build the full dump format.
  - Contains a main method to test the dumping format.
  - Implements Dumpable marker interface.
- **WriteCode:**
  - Takes no arguments to initialize
  - Simply just prints out the top of the runtime stack via requests to the virtual machine.
  - Implements Dumpable marker interface.

6.1   Class Diagram

- package interpreter.virtualmachine

- package interpreter.bytecode

ByteCodeLoader

CodeTable   Interpreter

(interpreter-bytecode-uml-diagram.png is in the documentation folder if this is difficult to see.)
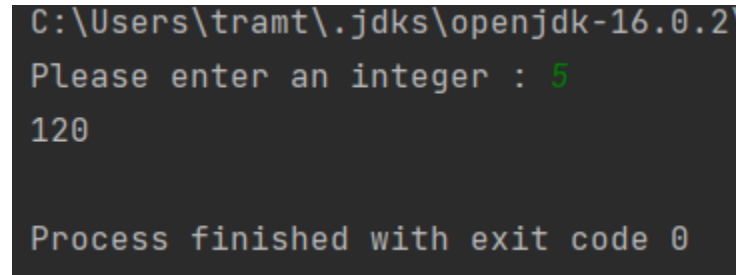
7 Project Reflection

This project is interesting, but also challenging at the same time. First off, there are no unit tests unlike project 1, so to debug I had to rely on the good old reliable print statements, but I also created main methods in certain classes to test their functions, like in the runtime stack or inside the bytecode classes that needed a specific dumping format. This project also required me to think of what the smallest or easiest things are to implement first, because I find it much easier to complete the harder implementations when I finished the simple ones first because I develop an idea of what each little function or class is supposed to do. I also enjoyed finding optimizations to code I had already written such as creating the Dumpable marker interface and abstract class BranchCode, to avoid creating static if checks as well as creating many conditionals and making it easier to reuse code that several bytecodes

have in common. Overall, this project was a good experience and thinking about how to make larger programs more maintainable for the next programmer or next new feature.

8 Project Conclusion/Results

In conclusion it appears to me that this project is working as intended because when I pass factorial.x.cod there should be nothing else printed to the console except the prompt and answer because looking inside the source code for factorial.x.cod we can see there is no DUMP ON command at all. I entered in 5 as an input for factorial and got back 120 which is the expected output of 5! = 120.


```
C:\Users\tramt\.jdks\openjdk-16.0.2\
Please enter an integer : 5
120


Process finished with exit code 0
```

When I entered in factorial.dump.cod with the same input 5, I also got 120 back as an answer and we can see each bytecode being dumped to the console correctly as well since Dump is turned on. (NOTE: the screenshots below do not show the full dump because screenshotting all of it will be difficult.)

```
C:\Users\tramt\.jdks\openjdk-16.0.2\bin\jav
[]
GOTO start<<1>>
[]
GOTO continue<<3>>
[]
ARGS 0
[] []
CALL Read        Read()
[] []
Please enter an integer : 5
READ
[] [5]
RETURN
[5]
ARGS 1
[] [5]
CALL factorial<<2>>      factorial(5)
[] [5]
LOAD 0 n         <load n>
[] [5, 5]
LIT 2
[] [5, 5, 2]
BOP <
[] [5, 0]
FALSEBRANCH else<<4>>
[] [5]
LOAD 0 n         <load n>
[] [5, 5]
LOAD 0 n         <load n>
[] [5, 5, 5]
LIT 1
```

```
120
WRITE
[] [120, 120]
RETURN
[120]
POP 3

[]
HALT
[]

Process finished with exit code 0
```

When using fib.x.cod we can see it does not have DUMP at all and the output should only produce a prompt and answer. I used a input of 10 which returns 55 which is the expected result.

```
C:\Users\tramt\.jdks\openjdk-16.0.2
Please enter an integer : 10
55

Process finished with exit code 0
```

I created a fib.dump.cod which simply has 'DUMP ON' as its first line to see all the byte codes that are being used.

```
[5, 55, 0, 7]
STORE 2 x          x=7
[5, 55, 7]
LIT 8
[5, 55, 7, 8]
STORE 2 x          x=8
[5, 55, 8]
POP 1

[5, 55]
POP 2

[]
HALT
[]

Process finished with exit code 0
```