VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FALCUTY OF COMPUTER SCIENCE AND ENGINEERING

Principle of Programming Languages - Extra class

Task 1

Mentor: Dr. Nguyen Hua Phung
MEng. Tran Ngoc Bao Duy
Student: Truong Nguyen Khoi Nguyen - 2010468

HO CHI MINH CITY, MARCH/2023

Contents

1 Theory

1.1 JVM

A Java virtual machine (JVM) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required in a JVM implementation. Having a specification ensures interoperability of Java programs across different implementations so that program authors using the Java Development Kit (JDK) need not worry about idiosyncrasies of the underlying hardware platform.

The JVM has two primary functions: to allow Java programs to run on any device or operating system (known as the "write once, run anywhere" principle), and to manage and optimize program memory. When Java was released in 1995, all computer programs were written to a specific operating system, and program memory was managed by the software developer. The JVM was a revelation.

JVM's architecture The components of the architecture:

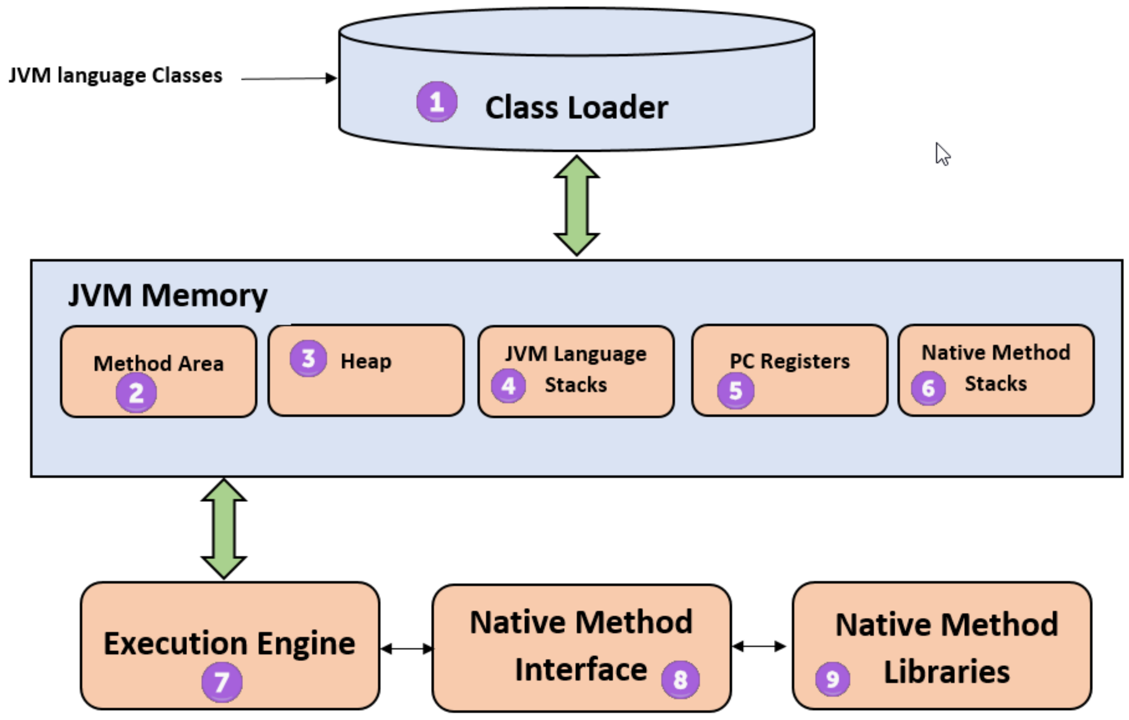


Figure 1: Internal JVM Architecture

1. **Class Loader:** The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.

2. **Method Area:** JVM Method Area stores class structures like metadata, the constant runtime pool, and the code for methods.

3. **Heap:** All the Objects, their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple threads.

4. **JVM Language Stacks:** Java language Stacks store local variables, and it's partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when method invocation process is complete.

5. **PC registers:** PC register store the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.

6. **Native Method Stacks:** Native method stacks hold the instruction of native code depends on the native library. It is written in another language instead of Java.

7. **Execution engine:** It is a type of software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.

8. **Native Method Interface:** The Native Method Interface is a programming framework. It allows Java code which is running in a JVM to call by libraries and native applications.

9. **Native Method Libraries:** Native Libraries is a collection of the Native Libraries(C, C++) which are needed by the Execution Engine.

1.2 Jasmin assembler

Jasmin is an assembler for the Java Virtual Machine. It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instruction set. It converts them into binary Java class files, suitable for loading by a Java runtime system.

Jasmin was originally created as a companion to the book "Java Virtual Machine", written by Jon Meyer and Troy Downing and published by O'Reilly Associates. Since then, it has become the de-facto standard assembly format for Java. It is used in dozens of compiler classes throughout the world, and has been ported and cloned multiple times. For better or worse, Jasmin remains the oldest and the original Java assembler.

To give you a flavor, here is the Jasmin assembly code for HelloWorld:

```
.class public HelloWorld
.super java/lang/Object

;
; standard initializer (calls java.lang.Object's initializer)
;
.method public <init>()V
   aload_0
   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

;
; main() - prints out Hello World
```

```
;
.method public static main([Ljava/lang/String;)V
   .limit stack 2   ; up to two items can be pushed

   ; push System.out onto the stack
   getstatic java/lang/System/out Ljava/io/PrintStream;

   ; push a string onto the stack
   ldc "Hello World!"

   ; call the PrintStream.println() method.
   invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

   ; done
   return
.end method
```

2 Code Generation Architecture

2.1 Code Generation Framework

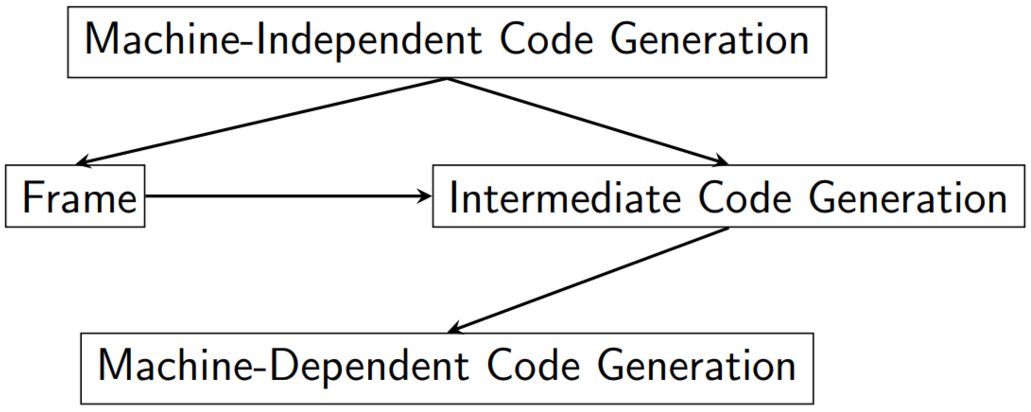

Machine-Independent Code Generation
Frame
Intermediate Code Generation
Machine-Dependent Code Generation

Figure 2: The framework for code generation

The code generation process is divided into 4 part, each one of them works with different purposes. The upper layer will use the API or class and method from the lower.

- Machine-Dependent Code Generation: This layer's purpose is creating all the instruction of the target assembly language such as Jasmin, MIPS, ... In initial file the components that relates to this layer is **MachineCode.py**.
- Intermediate Code Generation: All the API (Class and method) in this layer is created for user to easy to make the assembly code without knowing too much about the architecture of the bottom layer. This will help programmes who want to create

assembly code not spend their time to much when create JVM code. **Emitter.py** is the implementation of this stage in initial folder.

- Frame: Tools are used to manage information used to generate code for a method. The file that relates to this stage is **Frame.py**
- Machine-Independent Code Generation: This layer will use interfaces from Frame and Intermediate Code Generation to translate the AST tree of the input language to the target language. **CodeGenerator.py** is the represented of this stage in initial folder.

2.2 CodeGenerator Architecture

This is the main file that we want to focus during this task. The goal of this task is extended some files in initial folder to create JVM code (Jasmin) for Binary Expression (just the plus expression).

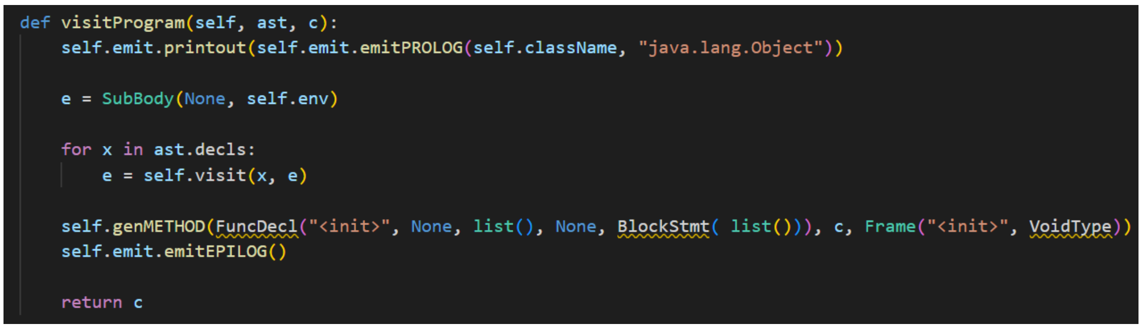
2.2.1 CodeGenerator Class

Inherit the Utils class. This class will create an object for code generation by visiting all the components of the AST tree.

2.2.2 CodeGenVisitor Class

Inherit from the BaseVisitor and Utils. Have some methods for visiting the AST tree's components and creating JVM code by using classes in Emitter.py and Frame.py

visitProgram Method: This visit method will map the Program function in MT22 to the MT22 Class in JVM.



```
def visitProgram(self, ast, c):
    self.emit.printout(self.emit.emitPROLOG(self.className, "java.lang.Object"))

    e = SubBody(None, self.env)

    for x in ast.decls:
        e = self.visit(x, e)

    self.genMETHOD(FuncDecl("<init>", None, list(), None, BlockStmt( list())), c, Frame("<init>", VoidType))
    self.emit.emitEPILOG()

    return c
```

Figure 3: VisitProgram Method

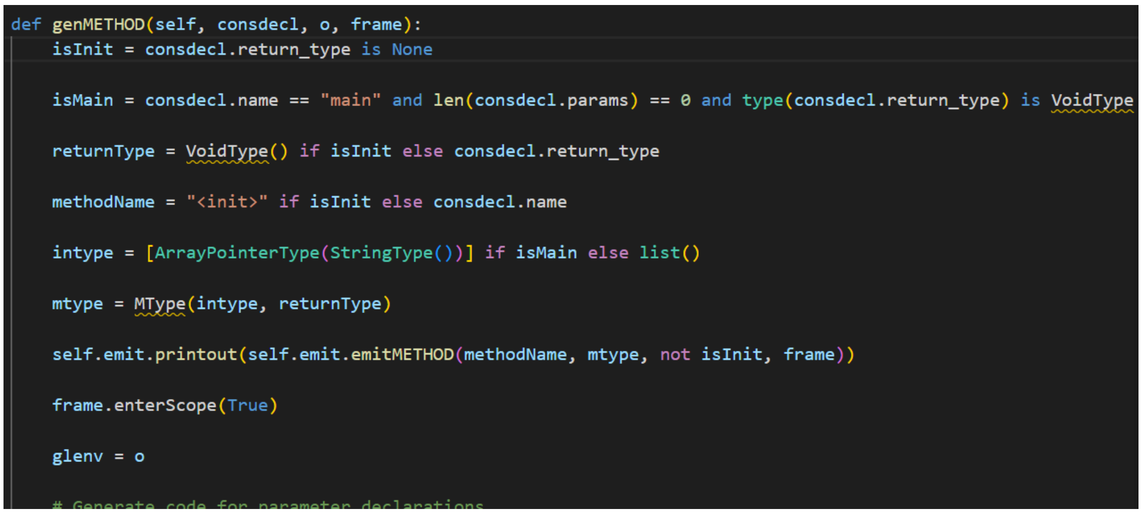
Input:

- ast: is the ast tree.
- c: the variable that holds the suitable environment for the visitProgram method, this variable will change due to the method.

Output: c

genMETHOD Method: This method is used for:

- Create the init method in JVM Code for the MT22 Class.
- Create the main method in JVM Code for the main function in MT22 Language.
- Create a user-specified method in MT22.



```
def genMETHOD(self, consdecl, o, frame):
    isInit = consdecl.return_type is None

    isMain = consdecl.name == "main" and len(consdecl.params) == 0 and type(consdecl.return_type) is VoidType

    returnType = VoidType() if isInit else consdecl.return_type

    methodName = "<init>" if isInit else consdecl.name

    intype = [ArrayPointerType(StringType())] if isMain else list()

    mtype = MType(intype, returnType)

    self.emit.printout(self.emit.emitMETHOD(methodName, mtype, not isInit, frame))

    frame.enterScope(True)

    glenv = o

    # Generate code for parameter declarations
```

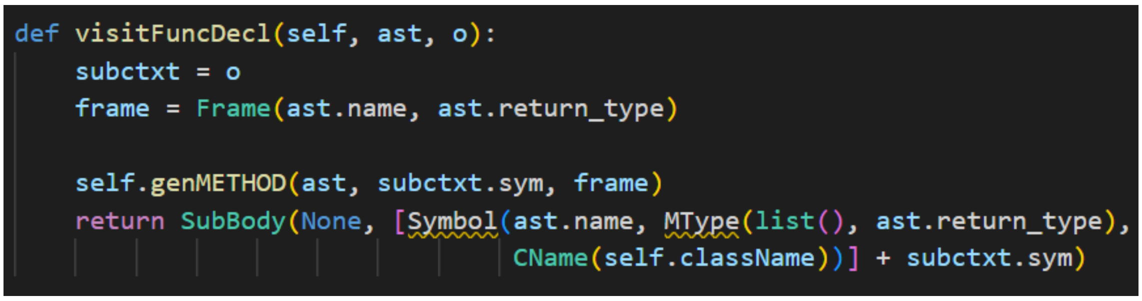
Figure 4: genMETHOD Method

Input:

- consdecl: the declarations that need to be mapped to JVM.
- o: The environment for this method. In this case, it is a list of Symbols for call statements if its exits.
- frame: The outside frame for this method

Output: No

visitFuncDecl Method: This will visit FuncDecl Class in the AST tree and then generate the JVM code by using the genMETHOD method above.



```
def visitFuncDecl(self, ast, o):
    subctxt = o
    frame = Frame(ast.name, ast.return_type)

    self.genMETHOD(ast, subctxt.sym, frame)
    return SubBody(None, [Symbol(ast.name, MType(list(), ast.return_type),
                                 CName(self.className))] + subctxt.sym)
```

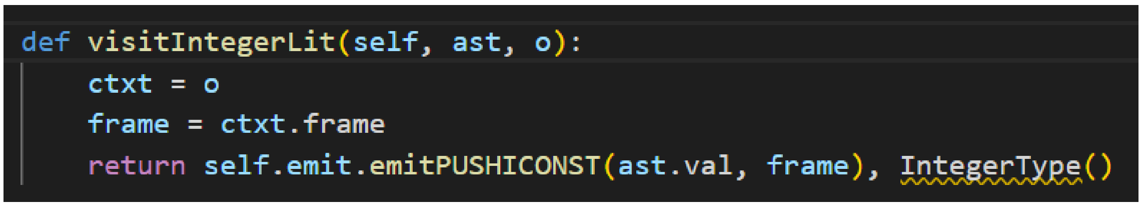
Figure 5: visitFuncDecl Method

Input:

- ast: the AST tree
- o: The environment for this method. In this case, it is a SubBody class of the outside function or class.

Output: No

visitIntegerLit method: This will visit the IntegerLit from the AST tree and then return the JVM code that pushes an integer on the top of the stack.



```
def visitIntegerLit(self, ast, o):
    ctxt = o
    frame = ctxt.frame
    return self.emit.emitPUSHICONST(ast.val, frame), IntegerType()
```

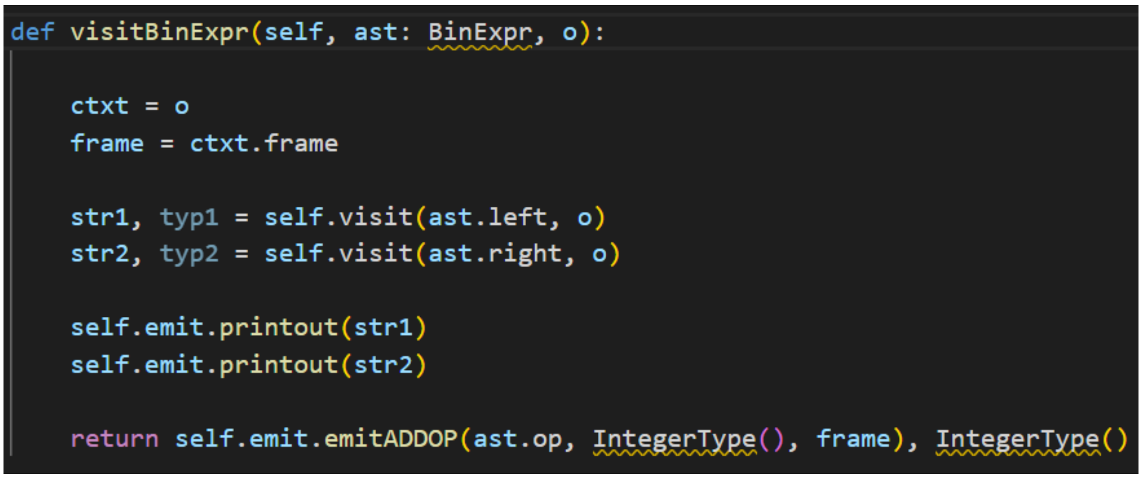
Figure 6: visitFuncDecl Method

Input:

- ast: the AST tree
- o: The environment for this method. In this case, it is an Access Class.

Output: The command that tells Emitter to create the push Integer on the top of the stack and the IntegerType

visitBinExpr method: This will visit the BinExpr from the AST tree then return the JVM code that simulates the process add 2 integer numbers in JVM.



```
def visitBinExpr(self, ast: BinExpr, o):

    ctxt = o
    frame = ctxt.frame

    str1, typ1 = self.visit(ast.left, o)
    str2, typ2 = self.visit(ast.right, o)

    self.emit.printout(str1)
    self.emit.printout(str2)

    return self.emit.emitADDOP(ast.op, IntegerType(), frame), IntegerType()
```

Figure 7: visitBinExpr Method

Input:

- ast: the AST tree
- o: The environment for this method. In this case, it is an Access Class.

Output: The command that tells Emitter to add two numbers and the IntegerType

This is the file that i craete for completing this task

1. First we need to assign the environment for this method

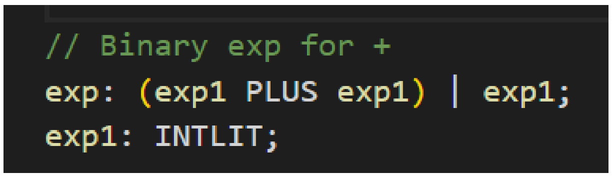
2. Then we traverse on the left and right expresssion of this AST to visit the emit method for these expression.

3. Thirdly, we use printout method to put these instruction into buffer and wait for the program write it to .j file

4. Finally, we create the add number (just for integer) function and set it as the return value of this method.

2.3 Addition changes in other file

2.3.1 MT22.g4

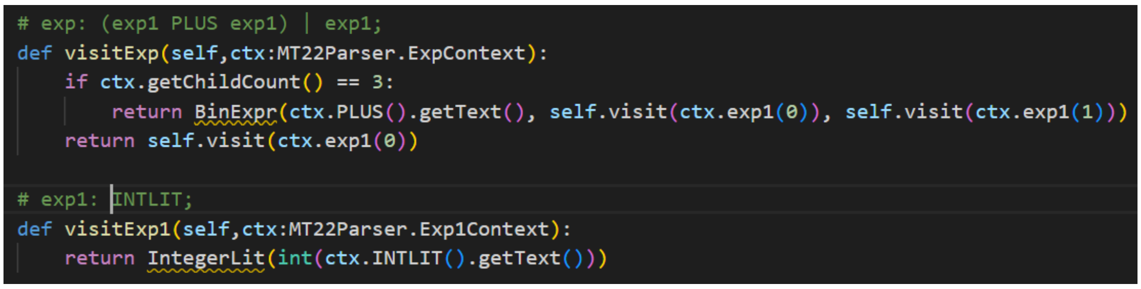


```
// Binary exp for +
exp: (exp1 PLUS exp1) | exp1;
exp1: INTLIT;
```

Figure 8: Changes in expr of MT22.g4 file

I extended the expression parser to create a binary expression instead of just 1 Integer Literals

2.3.2 ASTGeneration.py



```
# exp: (exp1 PLUS exp1) | exp1;
def visitExp(self,ctx:MT22Parser.ExpContext):
    if ctx.getChildCount() == 3:
        return BinExpr(ctx.PLUS().getText(), self.visit(ctx.exp1(0)), self.visit(ctx.exp1(1)))
    return self.visit(ctx.exp1(0))

# exp1: INTLIT;
def visitExp1(self,ctx:MT22Parser.Exp1Context):
    return IntegerLit(int(ctx.INTLIT().getText()))
```

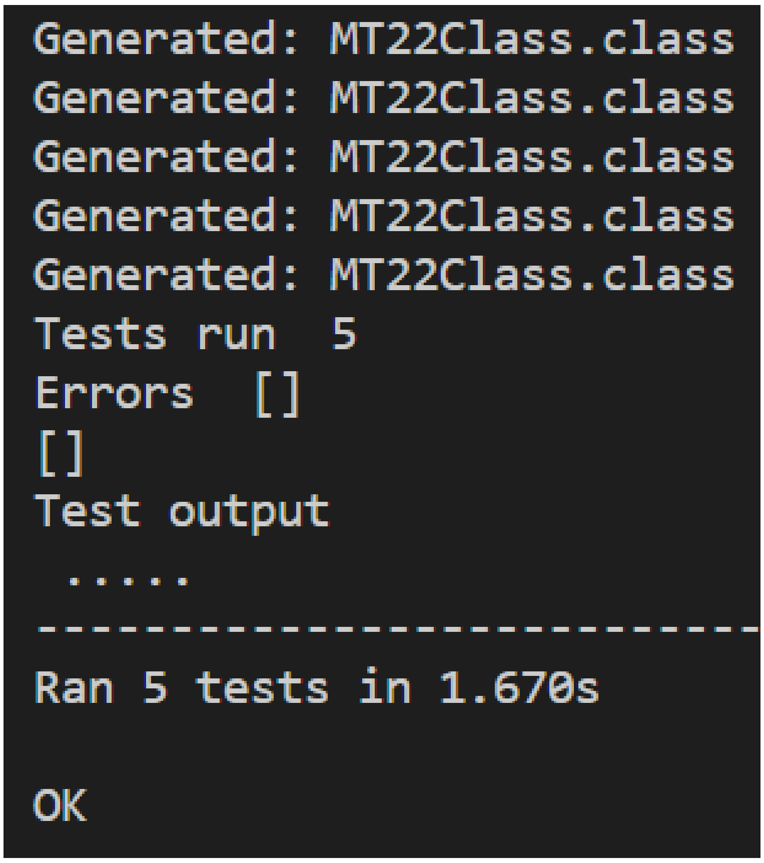
Figure 9: Changes in expr of MT22.g4 file

I also add the visit method for this file to create the AST from the token of lexer and parser suite.

3 Testing and result

I created 5 test function for this task, each testcase is a function putInt() with some plus expression with two integer as its argument

ALl the testcase are run and pass successfully



```
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Tests run  5
Errors  []
[]
Test output
 .....
---------------------------

Ran 5 tests in 1.670s

OK
```

Figure 10: Run result in command prompt

4 References

1. JASMIN HOME PAGE
2. CodeGeneration
3. JVM
4. What is JVM (Java Virtual Machine): Architecture Explained!