TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A

PROJECT REPORT

ON

A JVM IMPLEMENTATION FOR A SUBSET OF JAVA

SUBMITTED BY:

LOKESH PANDEY (PUL077BCT040)

MANDIP THAPA (PUL077BCT044)

MANISH KUNWAR (PUL077BCT045)

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING

September, 2024

Acknowledgments

This project is being undertaken as a course requirement of the Major Project, as a part of the final year curriculum of Bachelor in Computer Engineering (BCT).

First of all, we are greatly indebted to our project supervisor Asst. Prof. Jitendra Kumar Manandhar for his continuous guidance and supervision throughout the project.

We'd like to express our deep gratitude to our teachers and professors from the Department of Electronics and Computer Engineering (DoECE) for providing us with this opportunity to express our creativity and inspiring us to come up with efficient solutions to existing technological problems. The department has facilitated many students over the years with excellent infrastructure and curriculum .

We are sincerely thankful to our classmates for their diligent feedback and continuous motivation. Our appreciation goes to our seniors who have helped and inspired us to reach newer heights in programming pedigree.

Without the help and guidance of the aforementioned people, this study would not be possible.

Contents

List of Figures

List of Abbreviations

JVM	Java Virtual Machine
JIT	Just-in-Time
JRE	Java Runtime Environment
GC	Garbage Collection
MRE	Managed Runtime Environment
RAD	Rapid Application Development

1. Introduction

1.1 Background

A Java Virtual Machine (JVM) is a process virtual machine designed to run programs written in the Java programming language, as well as other languages that are compiled into Java bytecode. Its primary purpose is to provide a platform-independent runtime environment, allowing software developers to write code once and run it anywhere, irrespective of the underlying hardware or operating system. This cross-platform capability is a cornerstone of Java's "write once, run everywhere" philosophy. The JVM converts byte code written in Java into machine code/program that masks the specificities of the hardware and permits programmers to focus at and beyond the application level. Some of its major characteristics are automatic marring collection memory management, class loading on demand, several system resources management among others. As part of an octorpart in the software development process, JVM benefits also play an important aspect of making Java applications more secure and portable.

The purpose of this project is to create a tailored version of the Java Virtual Machine (JVM) that will execute a subset of the Java programming language. The designed JVM will be capable of reading and running class files that result from the compilation of java source files. The system will incorporate such essential components as exception handling, stack tracing, garbage collection to enhance the runtime experience. In this JVM, such basic features of the Java programming language as the use of primitive data types, arrays, and strings, control flow statements, classes, subclasses, interfaces, methods (virtual and static) are all present. Still, overhead features shall be left out such as reflection, multithreading, and Just-In-Time compilation (JIT) so as to keep the focus on the execution system. This project provides not only the understanding of the rationale behind every aspect of JVM design, but also the possibilities of building upon this and the need for research in the area of virtual machines.

1.2 Problem statements

- **Complexity of Existing JVM Implementations:**
  Existing JVM implementations are difficult to understand due to their broad feature sets and highly intricate internal architectures, making it challenging for beginners to grasp their underlying mechanisms.

- **Need for Simplified JVM for Learning Purposes:**
  There is a lack of simplified JVM implementations focused on core functionalities, which could serve as educational tools to help learners explore the foundational processes without being overwhelmed by advanced features.

- **Challenges in Bytecode Interpretation:**
  Understanding how bytecode is parsed and executed is complicated by the numerous processes embedded in modern JVMs, making it important to develop a system that focuses on the interpretation of a basic subset of Java features.

1.3 Objectives

- To develop a custom JVM capable of executing bytecode for a specific subset of Java language features.

- To gain a deeper understanding of how virtual machines function, including the processes of bytecode, memory management, and class loading.

- To analyze the engineering challenges and design decisions involved in the development of virtual machines.

1.4 Scope

- **Academic Study and Research:**
  This project will serve as an educational tool, allowing in-depth exploration and academic study of virtual machines, focusing on the core processes of bytecode interpretation, memory management, and exception handling. It will provide a foundational platform for students and researchers to understand the internal workings of a JVM without the complexity of advanced features like Just-In-Time (JIT) compilation and multithreading.

- **Comparison with Existing Frameworks:**
  While creating a simplified model of the JVM, this project aims to enable face to face comparison with several existing JVM frameworks such as Oracle HotSpot and OpenJ9. The comparison will focus on the differences that may be observed in feature sets, complexity, performance and design choices, thus showing the merits and demerits of full scope JVM implementations. It will, moreover, show how a more narrowed range may help in learning the basic functioning of a virtual machine.

2. Literature Review

2.1 Related work

Codename One[1], is an open-source cross-platform framework aiming to provide write once, run anywhere code for various mobile and desktop operating systems (like Android, iOS, Windows, MacOS, and others). It was created by the co-founders of the Lightweight User Interface Toolkit (LWUIT) project, Chen Fishbein and Shai Almog, and was first announced on January 13, 2012. It was described at the time by the authors as "a cross-device platform that allows you to write your code once in Java and have it work on all devices specifically: iPhone/iPad, Android, Blackberry, Windows Phone 7 and 8, J2ME devices, Windows Desktop, Mac OS, and Web. The biggest goals for the project are ease of use/RAD (rapid application development), deep integration with the native platform and speed." Codename One built upon the LWUIT platform abstraction by adding a simulator and a set of cloud-based build servers that build native applications from the Java bytecode.

Eclipse OpenJ9[2], is a high performance, scalable, Java virtual machine (JVM) implementation that is fully compliant with the Java Virtual Machine Specification. OpenJ9 can be built from source, or can be used with pre-built binaries available at the IBM Semeru Runtimes project for a number of platforms including Linux, Windows and macOS. OpenJ9 is also a core component of the IBM developer kit, which is embedded in many IBM middleware products, including WebSphere Application Server and Websphere Liberty. OpenJ9 is also a component of Open Liberty.

HotSpot[3], released as Java HotSpot Performance Engine, is a Java virtual machine for desktop and server computers, developed by Sun Microsystems which was purchased by and became a division of Oracle Corporation in 2010. Its features improved performance via methods such as just-in-time compilation and adaptive optimization. It is the de facto Java Virtual Machine, serving as the reference implementation of the Java programming language.

GraalVM[4], is a Java Development Kit (JDK) written in Java. The open-source distribution of GraalVM is based on OpenJDK, and the enterprise distribution is based on Oracle JDK. As well as just-in-time (JIT) compilation, GraalVM can compile a Java application ahead of time. This allows for faster initialization, greater runtime performance, and decreased resource consumption, but the resulting executable can only run on the platform it was compiled for. It provides additional programming languages and execution modes. The first production-ready release, GraalVM 19.0, was distributed in May 2019. The most recent

release is GraalVM for JDK 22, made available in March 2024.

Jikes Research Virtual Machine (Jikes RVM)[5], is a mature virtual machine that runs programs written for the Java platform. Unlike most other Java virtual machines (JVMs), it is written in the programming language Java, in a style of implementation termed meta-circular. It is free and open source software released under an Eclipse Public License.

2.2 Related theory

2.2.1 Process virtual machines

A process VM[6], sometimes called an application virtual machine, or Managed Runtime Environment (MRE), runs as a normal application inside a host OS and supports a single process. It is created when that process is started and destroyed when it exits. Its purpose is to provide a platform-independent programming environment that abstracts away details of the underlying hardware or operating system and allows a program to execute in the same way on any platform.

A process VM provides a high-level abstraction – that of a high-level programming language (compared to the low-level ISA abstraction of the system VM). Process VMs are implemented using an interpreter; performance comparable to compiled programming languages can be achieved by the use of just-in-time compilation.

This type of VM has become popular with the Java programming language, which is implemented using the Java virtual machine. Other examples include the Parrot virtual machine and the .NET Framework, which runs on a VM called the Common Language Runtime. All of them can serve as an abstraction layer for any computer language.

2.2.2 Stack Machine

Stack machine[7], is a computer processor or a virtual machine in which the primary interaction is moving short-lived temporary values to and from a push down stack. In the case of a hardware processor, a hardware stack is used. The use of a stack significantly reduces the required number of processor registers. Stack machines extend push-down automata with additional load/store operations or multiple stacks and hence are Turing-complete.

Most or all stack machine instructions assume that operands will be from the stack, and results placed in the stack. The stack easily holds more than two inputs or more than one result, so a rich set of operations can be computed. In stack machine code (sometimes called p-code), instructions will frequently have only an opcode commanding an operation, with no additional fields identifying a constant, register or memory cell, known as a zero address format.[8] A computer that operates in such a way that the majority of its instructions do not include explicit addresses is said to utilize zero-address instructions.[9] This greatly sim-

plifies instruction decoding. Branches, load immediates, and load/store instructions require an argument field, but stack machines often arrange that the frequent cases of these still fit together with the opcode into a compact group of bits. The selection of operands from prior results is done implicitly by ordering the instructions. Some stack machine instruction sets are intended for interpretive execution of a virtual machine, rather than driving hardware directly.

2.2.3 Parsing

Parsing, syntax analysis, or syntactic analysis is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term parsing comes from Latin pars (orationis), meaning part (of speech).
Within computational linguistics the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a data structure showing their syntactic relation to each other, which may also contain semantic information. Some parsing algorithms generate a parse forest or list of parse trees from a string that is syntactically ambiguous.[10]

2.2.4 Java class file

A Java class file is a file (with the .class filename extension) containing Java bytecode that can be executed on the Java Virtual Machine (JVM). A Java class file is usually produced by a Java compiler from Java programming language source files (.java files) containing Java classes (alternatively, other JVM languages can also be used to create class files). If a source file has more than one class, each class is compiled into a separate class file. Thus, it is called a .class file because it contains the bytecode for a single class.
File layout and structure:[11]

- **Magic Number:** 0xCAFEBABE
- **Version of Class File Format:** The minor and major versions of the class file.
- **Constant Pool:** Pool of constants for the class.
- **Access Flags:** For example, whether the class is abstract, static, etc.
- **This Class:** The name of the current class.
- **Super Class:** The name of the super class.
- **Interfaces:** Any interfaces implemented by the class.

- **Fields:** Any fields defined in the class.
- **Methods:** Any methods defined in the class.
- **Attributes:** Any attributes of the class (for example, the name of the source file, etc.).

2.2.5 Java bytecode

Java bytecode serves as the intermediary language for Java programs, acting as the instruction set for the Java Virtual Machine (JVM). Composed of compact single-byte instructions, bytecode enables cross-platform compatibility, allowing Java applications to run seamlessly on any system with a compatible JVM, without the need for source code compilation. This bytecode can be interpreted by the JVM or compiled into native machine code via just-in-time (JIT) compilation, facilitating efficient execution of Java applications across diverse hardware and software environments, thus underscoring its significance in achieving Java's platform independence and security objectives.[11]
At the core of Java bytecode lies a comprehensive instruction set architecture, encompassing various instruction types essential to Java's object-oriented programming model. The JVM operates as both a stack machine and a register machine, with method frames containing operand stacks and arrays of local variables. These components facilitate data manipulation, control transfer, object creation, method invocation, and other fundamental operations crucial to Java program execution. Each bytecode instruction, represented by a single byte opcode along with optional operands, contributes to the richness and versatility of Java bytecode, enabling the implementation of complex functionalities while maintaining efficiency and portability across different JVM implementations and hardware platforms.

2.2.6 Interpreter

An interpreter is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.[12] An interpreter generally uses one of the following strategies for program execution:

1. Parse the source code and perform its behavior directly;
2. Translate source code into some efficient intermediate representation or object code and immediately execute that;
3. Explicitly execute stored precompiled bytecode made by a compiler and matched with the interpreter's Virtual Machine.

2.2.7 Garbage Collection

Garbage collection (GC) is a type of autonomous memory management used in computer science.[13] Memory that was allocated by the program but is no longer referenced is referred to as garbage and is collected by the garbage collector. John McCarthy, an American computer scientist, created garbage collection in 1959 to make manual memory management in Lisp simpler.[14]

Programmers no longer have to manually manage memory by deciding which objects to deallocate, when to do so, and when to bring them back into the system thanks to garbage collection. Stack allocation, region inference, memory ownership, and their combinations are other comparable strategies. The amount of processing time devoted to garbage collection in a program can be substantial, which can negatively impact its performance.

3. Methodology

3.1 Feasibility Study

This feasibility study explores the prospects of developing "A JVM implementation for a Subset of Java". This study provides a comprehensive understanding of the project's potential, addressing challenges, and providing insights crucial for informed decision-making and successful implementation.

3.1.1 Technical Feasibility

The initial stage of this phase involved a comprehensive analysis of the complexity inherent in established JVM implementations, such as Oracle's HotSpot and IBM's OpenJ9. These JVMs incorporate advanced features including reflection, Just-in-Time (JIT) compilation, and multithreading, which significantly contribute to their complexity and high resource demands. In order to streamline the development process, we identified the necessity of excluding these advanced features and concentrating on the core JVM functionalities, namely bytecode interpretation, garbage collection, and fundamental exception handling. This narrowed scope was deemed technically feasible, as it reduced the architectural complexity, involved fewer subsystems, and resulted in a more lightweight and efficient virtual machine.

3.1.2 Resource Availability

The feasibility study also assessed the availability of programming tools and resources, including the Java Development Kit (JDK), libraries for parsing .class files. Resources available for research and documentation, such as official Java bytecode specifications and existing open-source JVMs, were reviewed for guidance on implementing key components.

3.1.3 Time and Skill Feasibility

Given the 10-month timeline for project completion, a detailed breakdown of tasks was established. The team's skill set in Java and system-level programming languages was considered sufficient for this scope, and no external training was deemed necessary. The plan was to dedicate the first 4 months to research and design, the next 4 months to implementation, and the 2 final months to testing and refinement.

3.2 Requirement Analysis

The requirement analysis was the most important part which helps in finding the exact deliverables (scope) and features of JVM. In this stage, the various requirements (functional and non-functional) were listed in order to have an implementation of a JVM which can handle few Java language features.

3.2.1 Functional Requirements

- **Class Loading:** Support class loading from various sources.
- **Bytecode Interpretation:** Interpret and execute Java bytecode instructions.
- **Method Execution:** Implement method invocation, return mechanisms, and exception handling.
- **Memory Management:** Handle dynamic memory allocation with garbage collection and stack management.
- **Class File Parsing:** Parse and verify Java class files conforming to the specification.
- **Exception Handling:** Ensure robust exception handling mechanisms.

3.2.2 Non-Functional Requirements

- **Performance:** Deliver high performance with low latency and high throughput.
- **Scalability:** Support large applications with extensive memory usage.
- **Portability:** Ensure compatibility across various hardware and operating systems.
- **Reliability:** Provide consistent execution and comprehensive error handling
- **Maintainability:** Design a modular, well-documented codebase for easy maintenance.
- **Extensibility:** Allow for future enhancements without major rewrites.

3.3 Working Principle

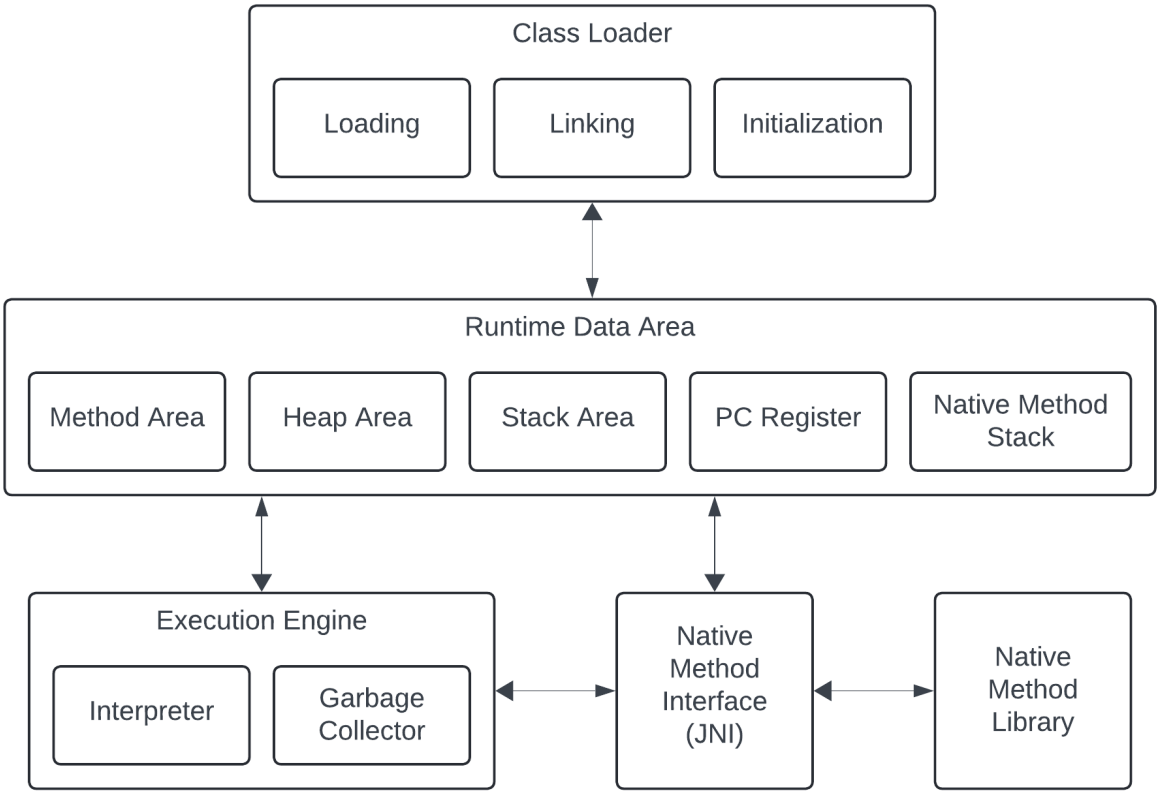

Figure 3.1: Architecture

3.3.1 Class loader

When you compile a .java source file, it is converted into byte code as a .class file. When you try to use the class in your program, the class loader loads it into the main memory. The first class to be loaded into memory is usually the class that contains the main() method. There are three phases in the class loading process: loading, linking, and initialization.

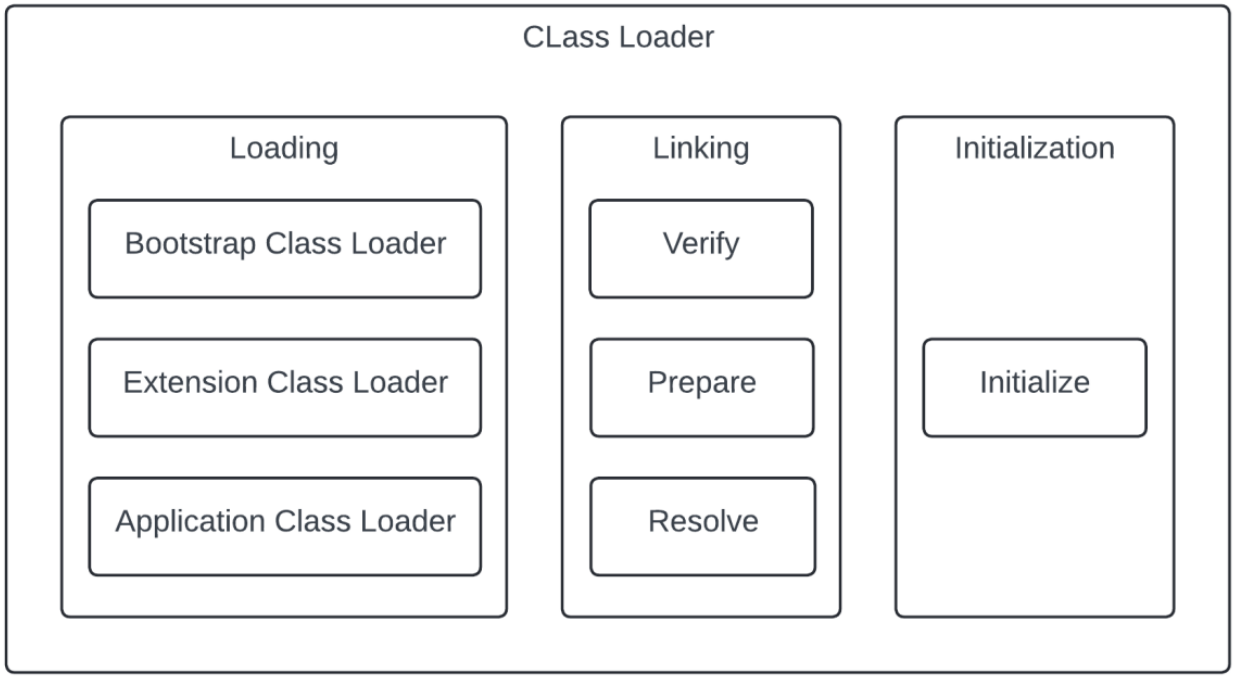

Figure 3.2: Class loader

Loading

Loading involves taking the binary representation (bytecode) of a class or interface with a particular name, and generating the original class or interface from that.
There are three built-in class loaders available in Java:

- **Bootstrap class loader** - This is the root class loader. It is the superclass of Extension Class Loader and loads the standard Java packages like java.lang, java.net, java.util, java.io, and so on. These packages are present inside the rt.jar file and other core libraries present in the $JAVA_HOME/jre/lib directory.
- **Extension class Loader** - This is the subclass of the Bootstrap Class Loader and the superclass of the Application Class Loader. This loads the extensions of standard Java libraries which are present in the $JAVA_HOME/jre/lib/ext directory.

- **Application class loader** - This is the final class loader and the subclass of Extension Class Loader. It loads the files present on the classpath. By default, the classpath is set to the current directory of the application.

Linking

After a class is loaded into memory, it undergoes the linking process. Linking a class or interface involves combining the different elements and dependencies of the program together. Linking includes the following steps:

- **Verification** - This phase checks the structural correctness of the .class file by checking it against a set of constraints or rules. If verification fails for some reason, exception is thrown.
- **Preparation** - In this phase, the JVM allocates memory for the static fields of a class or interface, and initializes them with default values.
- **Resolution** - In this phase, symbolic references are replaced with direct references present in the runtime constant pool.

Initialization

Initialization involves executing the initialization method of the class or interface. This can include calling the class's constructor, executing the static block, and assigning values to all the static variables. This is the final stage of class loading.

3.3.2 Runtime data area

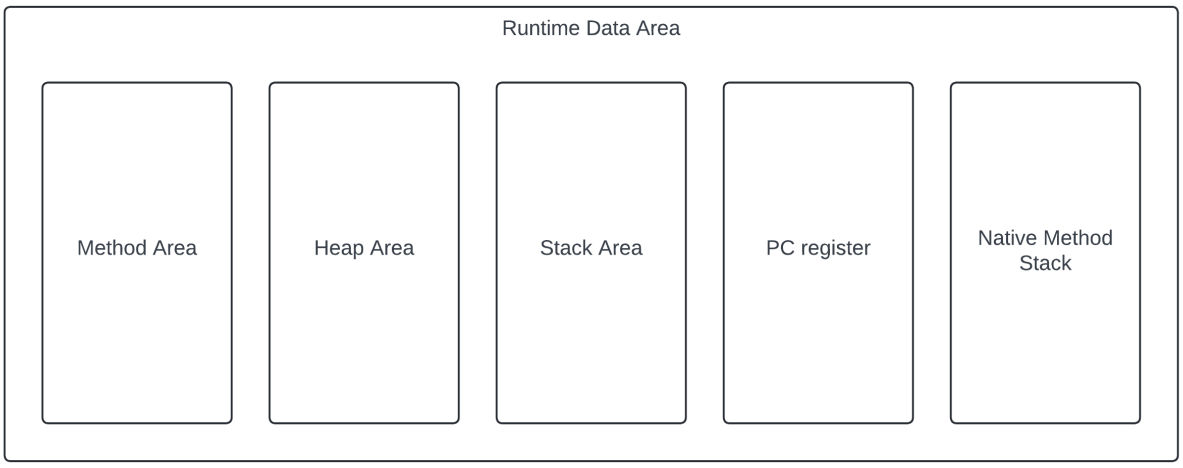

Figure 3.3: Runtime data area

Method area

All the class level data such as the run-time constant pool, field, and method data, and the code for methods and constructors, are stored here. If the memory available in the method area is not sufficient for the program startup, an exception is thrown. The method area is created on the virtual machine start-up, and there is only one method area per JVM.

Heap area

All the objects and their corresponding instance variables are stored here. This is the run-time data area from which memory for all class instances and arrays is allocated. The heap is created on the virtual machine start-up, and there is only one heap area per JVM.

Stack area

All local variables, method calls, and partial results are stored in the stack area. If the processing being done requires a larger stack size than what's available, an exception is thrown.
For every method call, one entry is made in the stack memory which is called the Stack Frame. When the method call is complete, the Stack Frame is destroyed.
The Stack Frame is divided into three sub-parts:

- **Local variables** - Each frame contains an array of variables known as its local variables. All local variables and their values are stored here. The length of this array is determined at compile-time.
- **Operand stack** - Each frame contains a last-in-first-out (LIFO) stack known as its operand stack. This acts as a runtime workspace to perform any intermediate operations. The maximum depth of this stack is determined at compile-time.
- **Frame data** - All symbols corresponding to the method are stored here. This also stores the catch block information in case of exceptions.

Program counter(PC) registers

PC Register holds the address of the currently executing JVM instruction. Once the instruction is executed, the PC register is updated with the next instruction.

Native method stack

The JVM contains stacks that support native methods. These methods are written in a language other than the Java, such as C and C++.

3.3.3 Execution engine

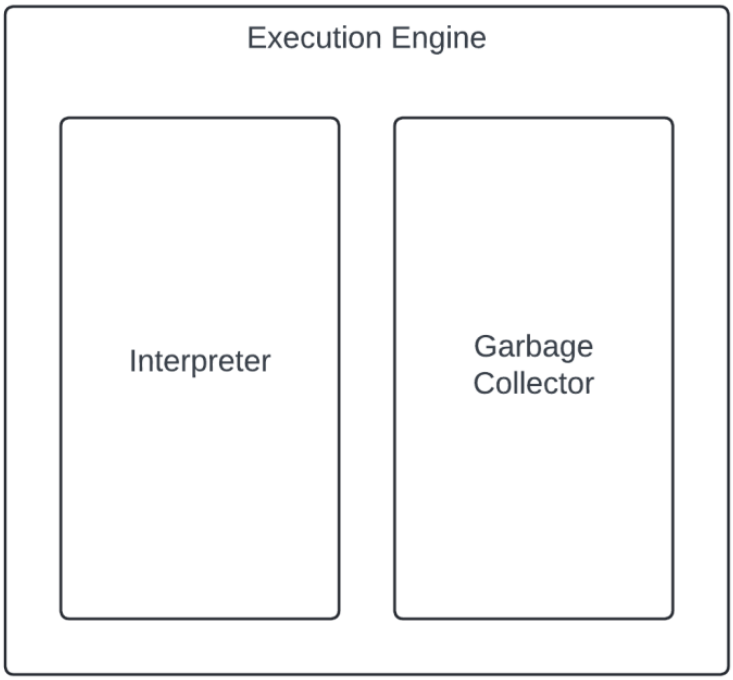

Figure 3.4: Execution engine

Interpreter

The interpreter reads and executes the bytecode instructions line by line. Due to the line by line execution, the interpreter is comparatively slower. Another disadvantage of the interpreter is that when a method is called multiple times, every time a new interpretation is required.

Garbage collector

The Garbage Collector (GC) collects and removes unreferenced objects from the heap area. It is the process of reclaiming the runtime unused memory automatically by destroying them. Garbage collection makes Java memory efficient because because it removes the unreferenced objects from heap memory and makes free space for new objects. It involves two phases:

1. **Mark** - in this step, the GC identifies the unused objects in memory
2. **Sweep** - in this step, the GC removes the objects identified during the previous phase.

Garbage Collections is done automatically by the JVM at regular intervals and does not need to be handled separately.

3.3.4 Java Native Interface(JNI)

At times, it is necessary to use native (non-Java) code (for example, C/C++). This can be in cases where we need to interact with hardware, or to overcome the memory management and performance constraints in Java. Java supports the execution of native code via the Java Native Interface (JNI).
JNI acts as a bridge for permitting the supporting packages for other programming languages such as C, C++, and so on. This is especially helpful in cases where you need to write code that is not entirely supported by Java, like some platform specific features that can only be written in C.

3.3.5 Native method libraries

Native Method Libraries are libraries that are written in other programming languages, such as C, C++, and assembly. These libraries are usually present in the form of .dll or .so files. These native libraries can be loaded through JNI.

4. System design

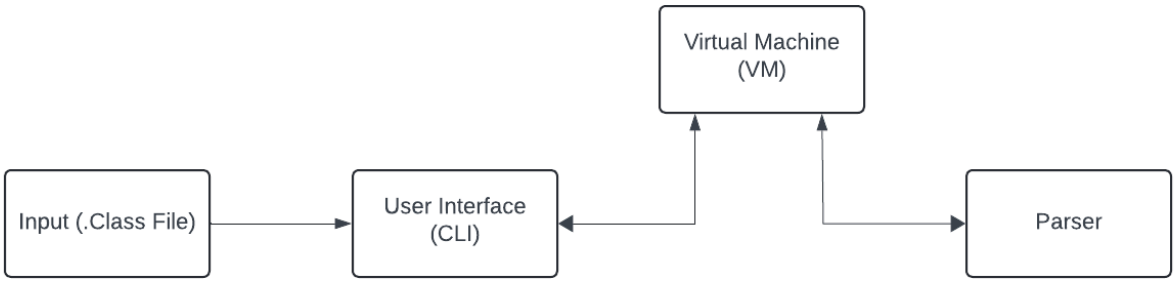

Figure 4.1: System Design

4.1 Components

4.1.1 Input(.Class File)

The .class file is the starting point of the JVM execution process. After Java source code is compiled, it is transformed into a .class file containing bytecode, a platform-independent set of instructions designed for execution by the JVM. This bytecode abstracts the complexities of various operating systems, making Java programs portable. When a .class file is supplied to the system, it is passed through the User Interface (CLI) for further processing by the virtual machine. This structure allows the JVM to interpret the bytecode and execute it in a way that is agnostic of the underlying hardware and software platform, ensuring "write once, run anywhere" functionality.

4.1.2 User interface (CLI)

The Command Line Interface (CLI) is the user-facing entry point for interacting with the JVM. It provides a means for users to supply inputs, such as class files, and manage JVM execution commands. Through the CLI, users load .class files, issue commands to execute them, and retrieve outputs or error messages from the JVM. Once a class file is provided, the CLI hands it off to the internal components of the JVM. This interface simplifies user interaction by abstracting the underlying complexities of bytecode loading, class management, and execution, presenting them in a command-line format that can be used in various environments.

4.1.3 Virtual Machine

The Virtual Machine (VM) is the heart of the JVM's architecture. It serves as the execution engine responsible for interpreting and running Java bytecode. The VM coordinates the entire process from loading classes, parsing bytecode, managing memory, and handling execution. It comprises several internal components, such as the Class Loader, Class Manager, Garbage Collector, execution stacks and interpreter, which work in tandem to ensure efficient program execution. The VM provides a consistent interface across platforms, ensuring that Java bytecode can run uniformly on any system equipped with a JVM. This modular approach within the VM allows it to manage resources like memory and provide features like security, and exception handling.

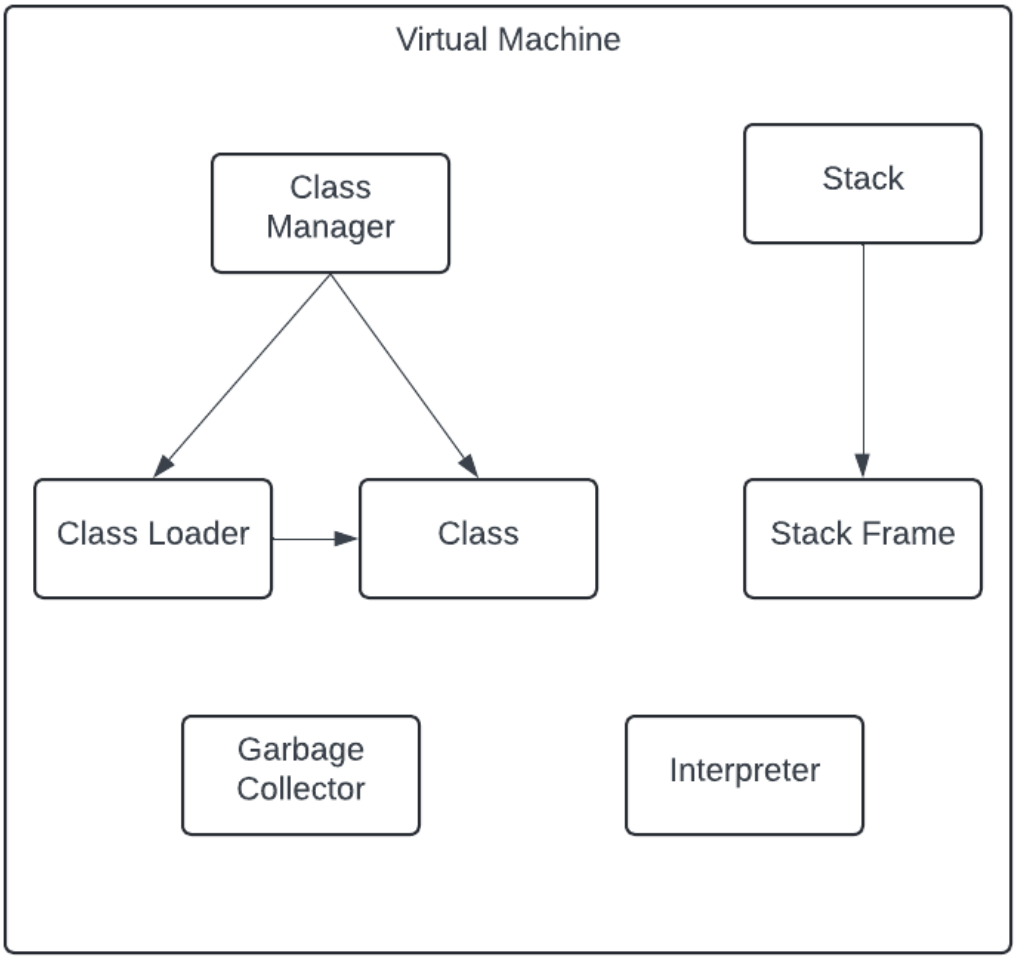

Figure 4.2: Component diagram of virtual machine

Class

A Class in the JVM represents a loaded class definition. It encapsulates the bytecode and metadata associated with the class, including its fields, methods, and constants. Once the class is loaded into memory, the JVM creates a Class object that holds all this information and can be referenced during execution. This Class object acts as a blueprint from which objects can be instantiated.

Class Loader

The Class Loader is responsible for dynamically loading classes into the JVM when they are needed. In Java, classes are loaded on demand, meaning they are not all loaded at the start of the program. The Class Loader follows a delegation model, where it first delegates the task of loading a class to its parent loader and only loads the class itself if the parent loader cannot find it. This hierarchical loading model ensures that classes are loaded efficiently and without duplication. The Class Loader allows the JVM to load classes from different sources, such as files, providing flexibility in how Java applications can be structured and run.

Class Manager

The Class Manager oversees the lifecycle of all Java classes loaded into the JVM. It is responsible for loading, linking, and initializing classes before they are executed. The Class Manager ensures that each class is loaded into memory correctly, linking them to other dependencies as needed and resolving symbolic references to actual memory addresses. It ensures that classes are only loaded once and manages the lifecycle of these class objects throughout their existence. The Class Manager works closely with other components, like the Class Loader and Garbage Collector, to keep track of loaded classes and ensure their proper utilization and memory management.

Stack

The Stack is a vital component of the JVM, managing the execution of methods by keeping track of method calls, local variables, and intermediate results. Every time a method is invoked, a new Stack Frame is created, which stores information related to that method's execution. This includes the parameters passed to the method, local variables, and the operand stack used for computations. The JVM's execution model is stack-based, meaning it pushes and pops data from the stack as methods are called and returned. The stack allows

the JVM to efficiently manage nested and recursive method calls, maintaining the execution context for each method invocation.

Stack Frame

A Stack Frame is created for each method invocation in the JVM. It contains all the necessary data required to execute the method, including local variables, parameters, intermediate results, and the return address. When a method is invoked, a new Stack Frame is pushed onto the Stack, and once the method completes, the frame is popped off, returning control to the previous method. This organization of frames allows the JVM to manage the execution of methods in a clean and structured manner, especially when dealing with recursive calls or methods that return values to their callers.

Interpreter

The Interpreter in the JVM is responsible for executing Java bytecode sequentially, one instruction at a time. It reads each bytecode instruction and directly translates it into native machine instructions, ensuring that the program is executed as intended. The interpreter handles method invocations, control flow, and expression evaluation by processing the bytecode step by step. This approach allows the JVM to execute Java programs without needing to convert the entire bytecode into native code upfront. Although it doesn't perform optimizations, the interpreter is essential for ensuring that Java applications run efficiently by directly interpreting and executing the bytecode in real-time.

Garbage Collector

The Garbage Collector is a key component responsible for automatic memory management in the JVM. Its job is to reclaim memory that is no longer in use by the program. As the program executes, objects that are no longer referenced by any part of the program are identified and marked for removal. The Garbage Collector then deallocates the memory occupied by these objects, making it available for future use. By automating memory management, the Garbage Collector prevents memory leaks and ensures that Java programs can manage resources efficiently without the need for manual intervention by the programmer. It operates in the background and is optimized to minimize its impact on program performance.

4.2 Language and Tools

4.2.1 Rust

Rust is particularly well-suited for building a JVM due to its strong emphasis on performance and memory safety. JVMs are complex systems that require efficient memory management, low-level access to system resources, and high concurrency support, all of which Rust excels at. Its ownership model eliminates common issues such as null pointer dereferencing and data races, making the system more robust and reliable. Rust also provides the control needed for fine-tuning the JVM's execution engine, garbage collector, and interpreter, while maintaining a balance between speed and safety. These features make Rust an ideal choice for developing a high-performance JVM with fewer runtime errors and more efficient memory usage.

4.2.2 OpenJDK

OpenJDK is utilized in the project to compile Java code into bytecode using the javac compiler, ensuring the bytecode adheres to Java language specifications. It serves as the reference implementation of the Java language, allowing the system to resolve and load superclasses during class execution. OpenJDK is crucial for verifying the correctness of the JVM's behavior, particularly in bytecode interpretation and class resolution, ensuring compatibility with existing Java applications.

5. Task Completed

5.1 Parser

The parser was developed to handle the extraction and processing of bytecode from .class files generated by compiling Java source code. This component is crucial as it reads the binary format of the Java class files and translates them into a more manageable data structure within the JVM. The parser decodes key information, such as constant pool entries, method definitions, fields, and attributes, ensuring that the bytecode can be understood and executed by the JVM. Special attention was given to properly handling Java's class file structure as defined in the Java Virtual Machine Specification, including handling complexities like references to methods, classes, and interfaces.

Listing 5.1: Data Structure to hold the Parsed Output

```
pub struct ClassFile {
    pub magic: U4,
    pub version: ClassVersion,
    pub constant_pool: Vec<ConstantInfo>,
    pub access_flags: U2,
    pub this_class: U2,
    pub super_class: U2,
    pub interfaces: Vec<U2>,
    pub fields: Vec<FieldInfo>,
    pub methods: Vec<MethodInfo>,
    pub attributes: Vec<AttributeInfo>,
}
```

5.2 Class Loader

The class loader was implemented to dynamically load classes during runtime. In traditional JVM implementations, the class loader follows the delegation model, where the loading of classes is passed up a hierarchy to prevent redundancy. For this JVM, a simplified class loading mechanism was created. It reads the .class file, resolves dependencies by loading any referenced classes (such as superclasses or interfaces), and links the bytecode to the runtime environment. This includes tasks like symbolic reference resolution, where names

and symbols in the bytecode are resolved to their actual memory addresses or runtime references, enabling smooth execution. The class loader plays a key role in ensuring that all necessary classes are available and properly loaded before execution.

Listing 5.2: Data Structure for Class

```
pub struct Class<'a> {
    pub id: ClassId,
    pub name: String,
    pub constants: ConstantPool,
    pub flags: U2,
    pub superclass: Option<ClassRef<'a>>,
    pub interfaces: Vec<ClassRef<'a>>,
    pub fields: Vec<FieldInfo>,
    pub methods: Vec<MethodInfo>,
    pub first_field_index: usize,
    pub total_fields: usize,
}
```

Listing 5.3: Data Structure for Class Loader

```
pub struct ClassLoader<'a> {
    classes: HashMap<String, ClassRef<'a>>,
}
```

Listing 5.4: Data Structure for Class Manager

```
pub struct ClassManager<'a> {
    class_path: ClassPath,
    class_by_id: HashMap<ClassId, ClassRef<'a>>,
    class_by_name: HashMap<String, ClassRef<'a>>,
    arena: Arena<Class<'a>>,
    next_id: u32,
    class_loader: ClassLoader<'a>,
}
```

5.3 Runtime Data Area

The Runtime Data Area is the memory structure used by the JVM during program execution. It includes the stack, which holds frames for method invocations, local variables, and the operand stack. The runtime data area also includes the method area (for class structure storage) and the PC register (to keep track of the current instruction being executed).

The memory structure for stack has been set up, ensuring proper memory allocation during execution. The stack is set up to handle method invocations and local variable storage. Each method invocation creates a new frame on the stack, which includes the local variables, operand stack, and a return value placeholder.

Listing 5.5: Data Structure for Stack Frame

```
pub struct StackFrame<'a> {
    locals: Vec<Value>,
    operand: Vec<Value>,
    cp: &'a ConstantPool,
    code: &'a Code,
    pc: U4,
}
```

Listing 5.6: Data Structure for Call Stack

```
pub struct Call_Stack<'a> {
    frames: Vec<StackFrame<'a>>,
}
```

6. Task Remaining

6.1 Execution Engine

6.1.1 Interpreter

The execution engine, specifically the interpreter, is currently under development. The interpreter is responsible for taking the bytecode instructions parsed from the .class files and executing them one by one. For this project, the interpreter is designed based on a stack-based architecture, meaning it operates by pushing operands onto a stack and then performing operations like addition, subtraction, method calls, etc., by popping the operands off the stack. The interpreter is being built to handle basic operations such as arithmetic calculations, method invocations, field access, control flow operations (like if, goto, and loops), and returning values from methods. Once fully developed, the interpreter will enable the JVM to run Java programs by continuously fetching bytecode instructions, interpreting them, and executing the corresponding low-level operations on the underlying hardware.

Implemented ByteCodes: Some Instructions required to test the Runtime Area like(Aaload, Aastore, Dload, Iload, Fload, Dstore, Istore, Fstore and so on) have already been implemented.

6.1.2 Garbage Collector

The garbage collector will handle automatic memory management by identifying and reclaiming memory used by objects that are no longer accessible by the program. The garbage collection algorithm to be implemented will be a basic mark-and-sweep system. During the "mark" phase, all objects that are still reachable (i.e., referenced by active parts of the program) are marked. In the "sweep" phase, any objects that are not marked are considered unreachable and their memory is reclaimed for future use. The garbage collector will also integrate with the heap to manage memory allocation and deallocation efficiently, preventing memory leaks and optimizing resource usage. This ensures that programs can run without manual memory management, adhering to Java's philosophy of automatic memory handling. Advanced features like generational garbage collection are not being implemented at this stage to keep the system focused on fundamental functionality.

6.2 Exception Handling

The exception handling system has yet to be implemented. This system will be responsible for managing runtime errors, such as divide-by-zero, null pointer dereferences, and array index out-of-bound errors. Exception handling in a JVM involves checking for error conditions during bytecode execution and throwing exceptions when necessary. When an exception is thrown, the JVM must look for the nearest appropriate handler in the method call stack, and if none is found, it will propagate the error up the stack, ultimately terminating the program if no handler is found. Additionally, this system will be tied to the stack trace mechanism, which will record method calls leading to the error and provide useful debugging information to the developer by printing the trace.

6.3 Verification

The verification process is essential for maintaining security and correctness before executing Java bytecode. Java's bytecode verifier ensures that the bytecode adheres to the JVM's strict constraints, preventing common errors like stack underflow/overflow or illegal data types being used. This is critical for avoiding security vulnerabilities, especially when running untrusted code. The verification process will include:

- **Data type checks:** Ensuring that operations are performed on the correct types.
- **Access control:** Verifying that code adheres to encapsulation and access rules, such as private or protected access.
- **Memory bounds checks:** Ensuring that array accesses and other memory operations stay within legal bounds. This step guarantees that code will not perform illegal or unsafe operations, making it a vital part of the JVM's security model.

References

[1] What is codename one. `https://shorturl.at/foZKb`, 2013. Archived from the original on October 24, 2012. Retrieved June 18, 2013.

[2] Leonardo Zanivan. New open source jvm optimized for cloud and microservices. `https://shorturl.at/868ik`, February 2018. Published on Medium.

[3] Sun announces availability of the java hotspot performance engine. Press Release, Sun Microsystems, 2013.

[4] Scott Lynn. For building programs that run faster anywhere: Oracle graalvm enterprise edition. Oracle Corporation, 2022. Retrieved 2022-01-21.

[5] The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2), 2005.

[6] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. Typed memory management in a virtual machine. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 287–303. ACM, 1996.

[7] Ian Jackson and John Earle. An efficient implementation of a stack machine. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 130–136. CSREA Press, 2000.

[8] Bob Beard. The kdf9 computer - 30 years on. *Computer RESURRECTION*, Autumn 1997.

[9] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[10] Masaru Tomita. *Generalized LR Parsing*. Springer Science & Business Media, December 2012.

[11] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *Java Virtual Machine Specification*, 202. Java SE 22 Edition.

[12] Mark D. McIlroy. The unix operating system and its influence on language design. *ACM SIGPLAN Notices*, 25(6):1–12, 1990.

[13] What is garbage collection (gc) in programming? `https://shorturl.at/uhMao`, 2024. Retrieved 2024-06-21.

[14] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960. S2CID 1489409. Retrieved 2009-05-29.