



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

March, 2025

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 sir for his continuous guidance and supervision throughout the project. Our supervisor has helped us tackle multiple facets of the project including helping us prepare our initial drafts, making sure we were on schedule for the project completion, providing us with research papers and contexts for tackling specific problems, evaluating our work progress in terms of output, presentation, and literature, guiding us in preparing this final report and, most importantly, helping us achieve the task of representing the campus as ideal students. We are eternally grateful.

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 and we are proud to represent our department to the best of our abilities.

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. The sincere and respectful culture of Pulchowk Campus is the best environment for fostering collaboration and competition among its students for novel projects and innovations in the years to come.

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

Abstract

In a landscape where traditional JVM implementations are complex and resource-intensive, this project presents a lightweight JVM designed to execute a subset of features of the Java language. Focused on education and research, it supports essential core functionalities such as class loading, bytecode interpretation, memory management, exception handling, and garbage collection, while excluding advanced features like Just-In-Time (JIT) compilation, multithreading, and reflection. Developed in Rust for performance and memory safety, the system includes a parser for Java class files, a class loader for dependency resolution, an execution engine for bytecode interpretation, and a basic garbage collector.

This project marks the culmination of four years of our engineering study, utilizing knowledge from various subjects such as operating systems and software engineering. Therefore, this project serves both facets of academia and practical implementation. By simplifying JVM internals while maintaining Java bytecode compatibility, this project provides a practical framework for understanding virtual machine architecture and lays the foundation for future enhancements.

Keywords: *Java Virtual Machine, Bytecode, Garbage Collection, Exception Handling, Platform Independence*

Contents

Acknowledgements	ii
Abstract	iii
Contents	v
List of Figures	vi
List of Abbreviations	vii
1 Introduction	1
1.1 Background	1
1.2 Problem statements	1
1.3 Objectives	2
1.4 Scope	2
2 Literature Review	3
2.1 Related work	3
2.2 Related theory	4
2.2.1 Process virtual machines	4
2.2.2 Stack Machine	4
2.2.3 Parsing	5
2.2.4 Java class file	5
2.2.5 Java bytecode	6
2.2.6 Interpreter	6
2.2.7 Garbage Collection	7
2.2.8 Generation Based Mark & Sweep Algorithm	7
2.2.9 Free List Management	8
3 Methodology	9
3.1 Feasibility Study	9
3.2 Requirement Analysis	9
3.2.1 Functional Requirements	10

3.2.2	Non-Functional Requirements	11
3.3	Working Principle	12
3.3.1	Parser	12
3.3.2	Class Loader	13
3.3.3	Interpreter	14
3.3.4	Memory Allocation	16
3.3.5	Garbage collection	19
3.3.6	Exception Handling	20
3.3.7	Native method stack	21
4	System design	22
4.1	Components	22
4.1.1	Input(.Class File)	22
4.1.2	User interface (CLI)	22
4.1.3	Parser	23
4.1.4	Virtual Machine	23
4.2	UML Diagrams	28
4.3	Language and Tools	34
4.3.1	Rust	34
4.3.2	OpenJDK	34
5	Result and Discussion	35
5.1	Normal Arithmetic and branching program	35
5.2	Inheritance program	36
5.3	Interface program	38
5.4	Polymorphism program	40
5.5	Multidimensional array program	42
5.6	Exception handling program	43
6	Conclusion	45
7	Limitations and Future Enhancements	46
7.1	Limitations	46
7.2	Future Enhancements	46
	References	48

List of Figures

3.1	Intermediate Representation	12
3.2	Heap structure	16
3.3	Free list structure	17
3.4	Garbage collection mechanism	19
4.1	System Design	22
4.2	Component diagram of virtual machine	23
4.3	Loaded class structure	24
4.4	Frame structure	26
4.5	State diagram for arithmetic instructions	28
4.6	State diagram for branching instructions	28
4.7	State diagram for method invocation	29
4.8	State diagram for exception handling	30
4.9	State diagram for memory allocation	31
4.10	State diagram for mark object	32
4.11	State diagram for garbage collection	33
5.1	Output of program to calculate the sum of even numbers upto 100	35
5.2	Output of program showcasing the inheritance	37
5.3	Output of program showcasing the interface	40
5.4	Output of program showcasing the polymorphism	41
5.5	Output of program showcasing the multidimensional array	43
5.6	Output of program showcasing the exception handling	44

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
IR	Intermediate Representation
OOP	Object-Oriented Programming
RTTI	Run-Time Type Information
CMS	Concurrent Mark-Sweep

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 memory collection, memory management, class loading on demand, several system resources management among others. As part of an important part 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:** 0xCAFEBAE
- **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.

2.2.8 Generation Based Mark & Sweep Algorithm

A garbage collection algorithm must carry out two essential tasks. First, it needs to identify all unreachable objects, and second, it must reclaim the heap space occupied by these garbage objects, making it available for the program once more. The Mark and Sweep Algorithm performs these operations in two phases, which are explained in more detail below:[15]

Phase 1: Mark Phase

When an object is created, its mark bit is set to 0(false). In the Mark phase, we set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true). Now to perform this operation we simply need to do a graph traversal, a depth-first search approach would work for us. Here we can consider every object as a node and then all the nodes (objects) that are reachable from this node (object) are visited and it goes on till we have visited all the reachable nodes.

- The root is a variable that refers to an object and is directly accessible by a local variable. We will assume that we have one root only.
- We can access the mark bit for an object by 'markedBit(obj)'.

Phase 1: Sweep Phase

As the name suggests it “sweeps” the unreachable objects i.e. it clears the heap memory for all the unreachable objects. All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to true. Now the mark value for all the reachable objects is set to false since we will run the algorithm (if required) and again we will go through the mark phase to mark all the reachable objects. The mark-and-sweep algorithm is called a tracing garbage collector because it traces out the entire collection of objects that are directly or indirectly accessible by the program.

- The garbage collector scans the entire heap.
- For each object encountered: If the object is marked, the mark is cleared (preparing it for the next GC cycle). If the object is not marked, it is considered garbage, and its memory is reclaimed (usually added to a free list for allocation to future objects).

2.2.9 Free List Management

In memory management, free list management is a technique used to track unallocated memory regions through a linked list structure. Each free memory block contains a pointer to the next, facilitating efficient dynamic memory allocation and deallocation. This approach is particularly effective in scenarios where memory allocations are of uniform size, such as in memory pool allocators, due to its simplicity and minimal overhead.

However, free list management has notable limitations. Over time, it can lead to memory fragmentation, resulting in numerous small, non-contiguous free blocks that may be insufficient for larger allocation requests. Additionally, allocating memory of a specific size may necessitate traversing the list to find a suitable block, which can be time-consuming as the list grows. To mitigate these issues, alternative methods like the buddy system allocator, which maintains multiple free lists for blocks of sizes that are powers of two, can be employed to reduce fragmentation and improve allocation efficiency.

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.

Technical Feasibility

The initial stage of this project involved a comprehensive analysis of established JVM implementations like HotSpot and OpenJ9, recognizing their complex architectures involving advanced features such as reflection, JIT compilation, and multithreading. To manage development complexity and resource constraints, we strategically decided to focus on core JVM functionalities including bytecode interpretation, basic memory management, and fundamental exception handling.

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.

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:** Class loading is the process of dynamically loading Java classes into memory as they are needed during program execution. It involves locating class files from various sources like file systems, network, or archives. The process includes three main phases: loading (finding and importing class definition), linking (verification, preparation, and resolution), and initialization (running static initializers).
- **Bytecode Interpretation:** Bytecode interpretation is the mechanism of converting Java bytecode into executable instructions that the computer can run. The JVM reads compiled Java bytecode and either interprets it directly or uses Just-In-Time (JIT) compilation to convert it to native machine code. This process ensures platform independence by translating standardized bytecode into instructions specific to the underlying hardware.
- **Method Execution:** Method execution manages how Java methods are invoked, executed, and completed. It involves creating and managing execution frames that track local variables, operand stacks, and method call information. The mechanism handles different types of method invocations, manages parameter passing, and supports complex control flow including exception handling.
- **Memory Management:** Memory management in JVM handles dynamic memory allocation and automatic garbage collection. It divides memory into different areas like heap (for objects), stack (for method invocations), and method area (for class metadata). The garbage collector automatically identifies and removes objects that are no longer referenced, preventing memory leaks and managing memory efficiently.
- **Class File Parsing:** Class file parsing involves reading and validating Java class files according to the JVM specification. It checks the structural integrity of the class file, verifies the bytecode, and extracts necessary metadata about classes, methods, and fields. This process ensures that only valid and safe class files are loaded and executed.

- **Exception Handling:** Exception handling provides a robust mechanism for detecting, reporting, and managing runtime errors. It allows the JVM to catch and process unexpected conditions, propagate error information through the call stack, and provide a mechanism for graceful error recovery. The system ensures that exceptions are properly traced, logged, and can be handled by appropriate catch blocks.
- **Garbage Collection:** Garbage Collection (GC) is an automatic memory management mechanism that identifies and eliminates unused objects from the heap memory. It tracks object references, determines which objects are no longer accessible, and frees the memory they occupy. Different GC algorithms are employed to optimize memory usage, minimize application pause times, and prevent memory leaks, ultimately improving overall application performance and reliability.

3.2.2 Non-Functional Requirements

- **Performance:** The JVM must minimize execution overhead and maximize computational efficiency. It should optimize bytecode interpretation and provide adaptive compilation techniques to reduce runtime performance penalties.
- **Scalability:** The implementation must handle memory-intensive applications with dynamic resource allocation. It should efficiently manage memory and support large-scale applications through intelligent memory management strategies.
- **Portability:** The JVM should provide a consistent runtime environment across different hardware architectures and operating systems. It must abstract platform-specific details to ensure "Write Once, Run Anywhere" principle.
- **Reliability:** The system must ensure consistent and predictable execution under various computational conditions. It should implement comprehensive error detection, logging, and recovery mechanisms to prevent application crashes.
- **Maintainability:** The JVM codebase should be modular, well-structured, and extensively documented for easy understanding and modification. It must follow clean code principles and provide clear separation of concerns.
- **Extensibility:** The architecture should support easy integration of new features and optimization techniques without fundamental redesign. It must provide plugin mechanisms and flexible interfaces for future enhancements.

3.3 Working Principle

3.3.1 Parser

The parser is the first step in the JVM pipeline, tasked with interpreting and validating the bytecode in Java class files, converting it into an intermediate representation (IR) for further processing.

- **Bytecode parsing:** The parser processes .class files, which follow a rigid binary format defined by the Java Virtual Machine Specification. These files contain metadata, a constant pool, method definitions, and bytecode instructions. The parsing begins by validating the magic number (0xCAFEBAE) and ensuring compatibility with supported class file versions.
- **Constantpool resolution:** The constant pool is a repository of symbolic references such as class names, method signatures, and literals. During parsing, these symbolic references are resolved into Rust-compatible data structures, ensuring efficient subsequent access.
- **Intermediate Representation (IR):** The IR is designed to provide a structured and hierarchical representation of the class file content. It includes details of methods, fields, attributes, and bytecode instructions, making it easier to feed into the class loader and interpreter.

magic: U4 (32-bit)
version: U4 (32-bit)
constant_pool: Variable size
access_flags: Variable size
this_class: U2 (16-bit)
super_class: U2 (16-bit)
interfaces: Variable size
fields: Variable size
methods: Variable size
attributes: variable size

Figure 3.1: Intermediate Representation

3.3.2 Class Loader

The class loader dynamically loads and links classes at runtime, ensuring they are initialized before usage.

- **Loading phase:** The class loading phase in the Java Virtual Machine (JVM) involves loading class files into memory for execution. The process begins with the Bootstrap Class Loader, which is responsible for loading core Java classes, such as `java.lang.Object`, and system libraries that are essential for the JVM's operation. It searches predefined locations, such as the `CLASSPATH` environment variable or embedded JAR archives, to locate these classes. The Bootstrap Class Loader operates at the highest priority in the class loading hierarchy.
- **Linking phase:** The linking phase in class loading involves transforming a loaded class into a fully usable form. This phase is responsible for preparing the class's structure in memory by resolving symbolic references and ensuring that all static fields and methods are properly associated with their actual memory locations. It includes both the preparation and resolution steps, ensuring that the class is ready for execution.
 - **Preparation:** Static fields are allocated memory and initialized to default values.
 - **Resolution:** Converts symbolic references in the constant pool into direct references to methods, fields, or classes in memory.
- **Initialization Phase:** The initialization phase of class loading involves the execution of static initializers (`<clinit>` methods) to initialize static fields and run any static code blocks within the class. This ensures that all static variables are set up correctly before the class is used by any thread. The static initializers are executed when the class is first accessed, ensuring proper initialization of the class's static state. To maintain thread safety during this process, the JVM employs synchronization mechanisms. This ensures that only one thread can initialize a class at a time, even if multiple threads attempt to load the class concurrently, preventing race conditions and guaranteeing that static fields are initialized in a consistent and safe manner.
- **Class Loader cache:** The class loader cache plays a crucial role in optimizing the class loading process by storing already-loaded classes in a hashmap. This prevents redundant loading of the same class, thereby improving runtime efficiency. When a class is requested, the class loader first checks the cache to see if the class has already been loaded. If it is present, the cached version is used, saving the time and resources required for reloading. This mechanism also ensures that each class is loaded only

once, maintaining uniqueness and consistency within the JVM, and preventing issues that could arise from loading multiple instances of the same class.

3.3.3 Interpreter

The interpreter executes Java bytecode instructions sequentially, simulating a virtual stack machine.

- **Execution model:** The interpreter follows a fetch-decode-execute cycle, repeatedly fetching bytecode instructions, decoding their opcodes, and executing the corresponding operations.
- **Runtime Data Area:** Each method call results in the creation of a new stack frame within the stack. A stack frame represents the execution context for a method, holding everything needed to execute the method and manage its state during execution.

The stack frame contains several key components:

- **Local variable array:** This array stores the method's arguments and local variables. It holds primitive data types and object references, with each variable having an index for quick access. Local variables are allocated when the method is called and deallocated when the method returns.
- **Operand stack:** This stack is used for holding intermediate values during method execution. It serves as a temporary storage space for operations like arithmetic, method calls, and returning values. Values are pushed onto the operand stack for calculations and popped off when results are needed.
- **Reference to runtime constant pool:** Each stack frame contains a reference to the constant pool of the defining class. The constant pool holds symbolic references to constants, field names, method names, and other class-level data that the method might need. The pool allows the JVM to resolve references to fields and methods during execution.
- **Program Counter (PC) register:** The PC register holds the address of the next instruction to be executed in the method. It helps the JVM keep track of which instruction in the method's bytecode is currently being executed. This ensures proper sequencing and flow control within the method.
- **Series of instructions (bytecode):** Each method is associated with a series of bytecode instructions that need to be executed. These instructions are stored in the method's bytecode array and are executed sequentially. The PC register

points to the current instruction in this series, and as each instruction is executed, the PC is updated to point to the next instruction.

- **Bytecode execution:** The bytecode execution, collectively enable the JVM to interpret and execute Java programs, handling arithmetic operations, control flow, object manipulation, and exception management efficiently in a stack-based environment.
 - **Arithmetic operations:** Bytecode instructions related to arithmetic operations handle both basic and complex calculations using the operand stack. The operand stack temporarily holds values that are involved in operations such as addition, subtraction, multiplication, and division. For instance, when an arithmetic operation is performed, operands are popped from the stack, the operation is executed, and the result is pushed back onto the stack.
 - **Control flow:** Control flow instructions in bytecode manage the execution order of instructions and enable branching logic. These include operations like goto and if statements. These control flow mechanisms are essential for implementing loops, conditional statements, and method invocations. When a branch is encountered, the Program Counter (PC) register is updated to point to the target instruction, ensuring that execution continues at the correct location based on the flow of the program.
 - **Object instructions:** Bytecode also includes instructions that facilitate object-oriented programming operations, such as object creation, method invocation, and field access. For object creation, the new instruction is used to instantiate objects by allocating memory for them on the heap. The invokevirtual instruction is used to invoke instance methods on objects, dynamically resolving the method to be called based on the object's actual type (polymorphism). Additionally, bytecode instructions like getfield and putfield are used to access or modify object fields.
 - **Exception handling:** Exception handling in bytecode is crucial for managing errors and abnormal program termination. Each method has an associated exception table, which contains information about which exceptions can be thrown and where the corresponding handlers are located. When an exception occurs, the JVM consults this table to find the appropriate handler. Bytecode instructions like athrow trigger the exception-handling mechanism, transferring control to the relevant catch block or method to manage the error.
- **Optimization:** Frequently executed methods are enhanced to improve performance. One common technique used here is caching, where the frequently used methods are

stored for quick retrieval, avoiding redundant computation. This optimizations reduce execution time, making the JVM more efficient and responsive during program execution.

3.3.4 Memory Allocation

Memory allocation is managed using a free-list-based mechanism, which is implemented with a double-linked list. This structure allows for efficient tracking and management of unallocated memory blocks within the heap, facilitating quick allocations and deallocations while minimizing fragmentation.

- **Heap management:** The heap is organized into different generations to optimize garbage collection (GC). Typically, this approach divides the heap into:
 - **Young generation:** This area contains recently allocated objects that are expected to die young (i.e., they will be garbage collected quickly).
 - **Old generation:** Objects that have survived multiple garbage collection cycles are promoted to this region. These objects tend to live longer.

This generational approach helps improve GC efficiency by focusing on frequent, small collections in the Young generation while minimizing the collection frequency in the Old generation.

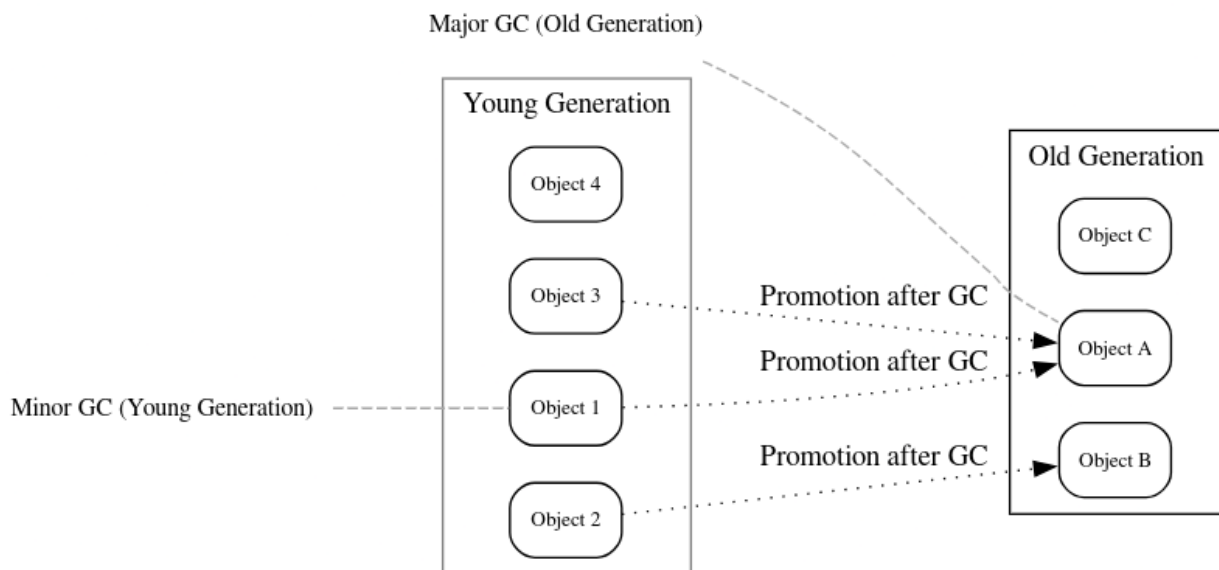


Figure 3.2: Heap structure

- **Free list structure:** The free list tracks unallocated memory blocks using nodes, each containing metadata to manage and organize the free space:
 - **Next pointer(next):** A reference to the next free block in the list.
 - **Previous pointer(prev):** A reference to the prev free block in the list.

This doubly linked structure allows for efficient traversal of the free list in both directions, enabling faster insertions and deletions of free blocks.

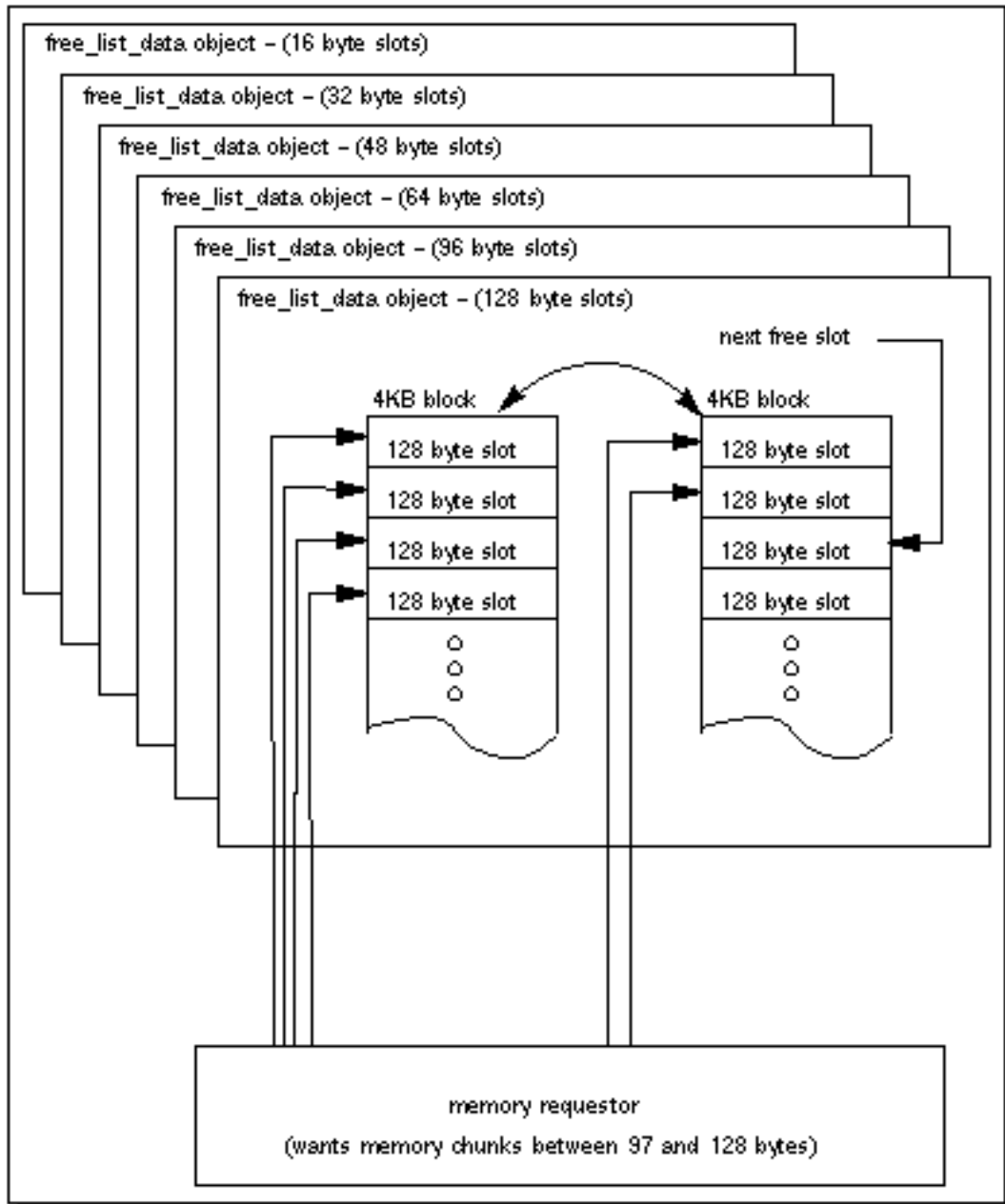


Figure 3.3: Free list structure

- **Allocation mechanism:** When a memory allocation request is made, the allocator traverses the free list to find a suitable free block. The allocation process typically follows these steps:
 - **Search the free list:** The allocator scans through the free list to find a block large enough to fulfill the memory request.
 - **Block selection:** Once a fitting block is found, it is either allocated entirely or split if it's larger than required. If the block is larger than needed, the remaining portion of the block is returned to the free list as a new, smaller free block.
 - **Block allocation:** The selected block is then marked as allocated, and its meta-data is updated accordingly to reflect the allocation.
- **Thread safety:** To ensure that memory allocation and deallocation are safe in multithreaded environments, the system uses mutexes to ensure that only one thread can access the free list at a time, preventing race conditions when allocating or freeing memory.
- **Deallocation:** When an object is no longer referenced and is eligible for garbage collection, the allocator reclaims the associated memory:
 - **Marking the block as free:** The memory block is updated to reflect that it is now free and available for future allocations.
 - **Returning to free list:** The reclaimed memory block is added back to the free list.

3.3.5 Garbage collection

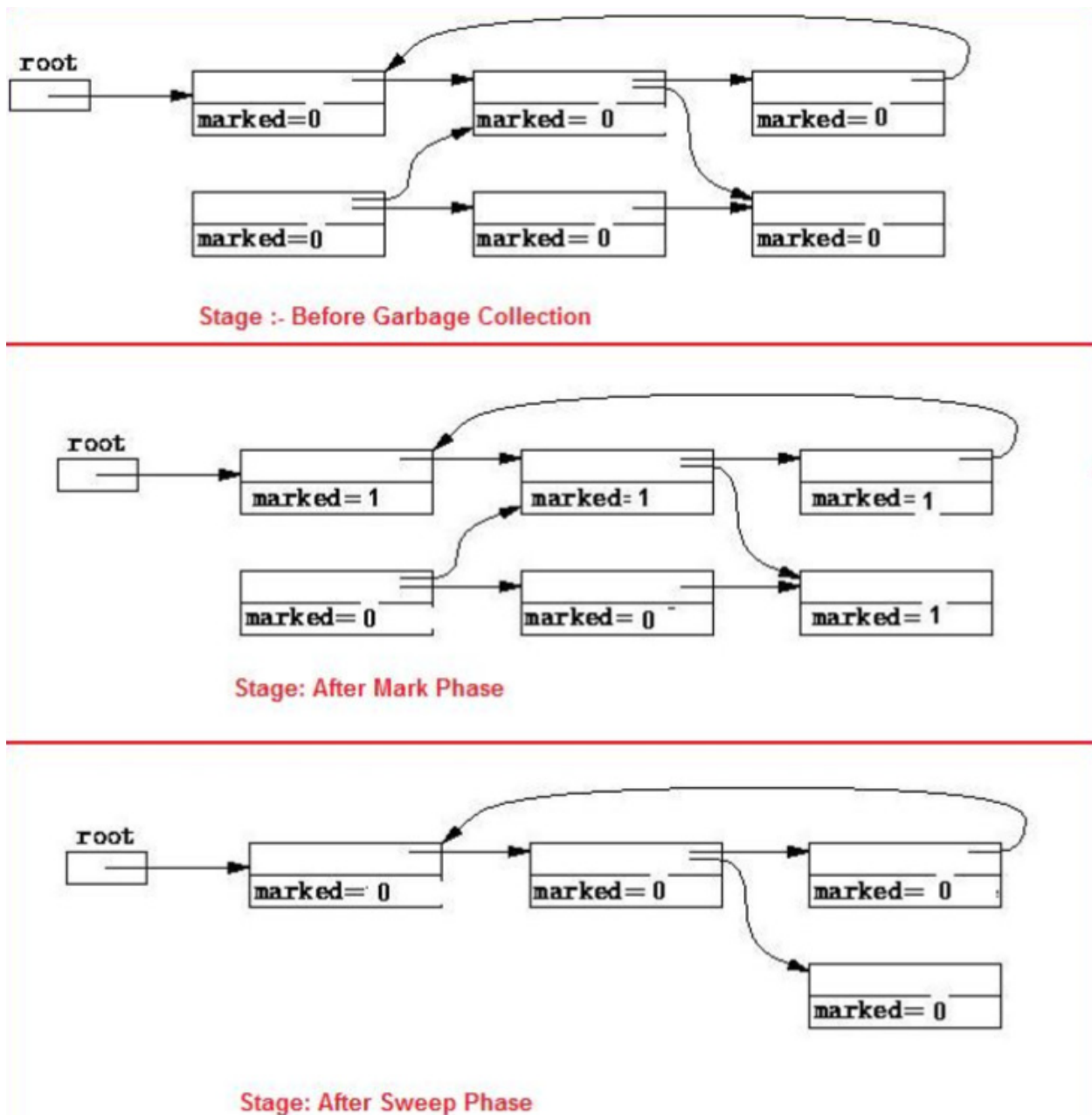


Figure 3.4: Garbage collection mechanism

A generation-based mark-and-sweep garbage collector is employed to manage heap memory efficiently. It divides memory into different generations to optimize garbage collection (GC) and reduce fragmentation over time.

- **Mark phase:** In the Mark Phase, the collector starts from root references (such as static fields, stack variables, and registers) and traverses the object graph to mark all objects that are reachable (i.e., still in use). This ensures that objects which are still

referenced are not mistakenly freed.

- **Sweep phase:** During the Sweep Phase, the collector identifies all unreachable objects that were not marked in the mark phase. These objects are considered garbage and their memory is reclaimed, returning it to the free list for future allocations. In the Old Generation, objects are compacted after being swept to reduce fragmentation. This involves moving objects closer together, which creates larger contiguous blocks of free memory, improving memory allocation efficiency.
- **Concurrency:** To minimize GC pause times, background threads are employed to perform marking and sweeping concurrently with the application's execution. This approach reduces the impact of garbage collection on the overall performance of the application, allowing most of the work to be done in parallel with the program's operation.
- **Exception Handling:** During garbage collection, references in stack frames (such as those for exception objects or handlers) are carefully preserved to ensure that any exceptions being thrown or caught remain valid and that the program can resume execution correctly. This guarantees that garbage collection does not interfere with the handling of exceptions.

3.3.6 Exception Handling

The JVM provides a robust mechanism for handling exceptions, ensuring that programs can recover from runtime errors and maintain control flow even when unexpected conditions occur.

- **Try-Catch Block**
 - **Exception table:** Metadata in the compiled .class file includes an exception table, which maps different types of exceptions to the corresponding catch blocks. This allows the JVM to quickly identify where to jump in case an exception is thrown.
 - **Try catch mechanism:** When an exception occurs, the JVM checks the exception type against the entries in the exception table, directing the flow to the appropriate catch block for handling.
- **Throwing exception:** When an exception is thrown, the JVM begins stack unwinding. It pops method frames off the stack one by one until it finds a catch block that matches the exception type.

If no matching catch block is found in the current method or its callers, the exception is passed up to the caller's method, continuing the unwinding process until a handler is found or the program terminates.

- **Synchronization with frames:** During stack unwinding, the JVM ensures that method frames are properly cleaned up. All local variables and references within a frame are destroyed to free memory.

The JVM also ensures that any references that could lead to memory leaks are cleaned up, preventing issues like dangling references during exception handling.

3.3.7 Native method stack

The .dll or .so library file obtained through the compilation of C/C++ is dynamically loaded at runtime. From these libraries, the necessary native functions are retrieved and invoked. This mechanism bridges the gap between Java bytecode and platform-specific functionalities, enabling seamless execution of native operations. The loading process ensures compatibility with the underlying hardware and operating system, while the execution integrates these native functions into the JVM workflow without compromising thread safety or runtime efficiency.

4. System design

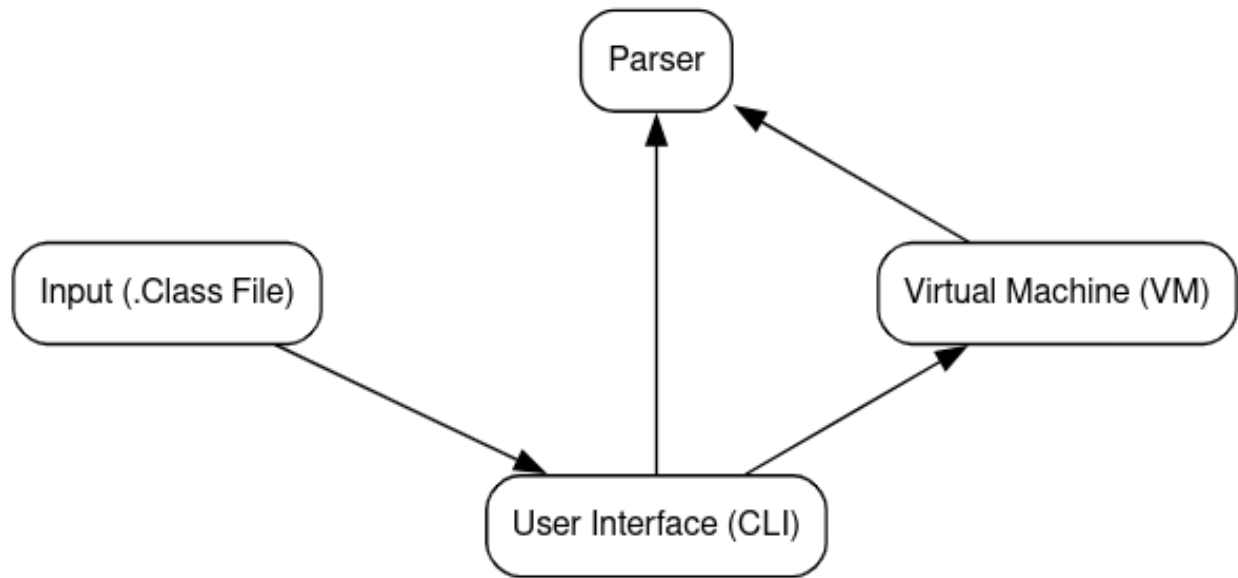


Figure 4.1: System Design

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 Parser

The Parser processes the bytecode from ‘.class’ files, translating it into a format that the JVM can execute. It reads the bytecode, and extracts method information, constants, and metadata. The parser resolves references to methods, fields, and constants, linking them to appropriate runtime values or memory locations. This enables the JVM to interpret and execute the Java program by converting bytecode instructions into executable actions, ensuring the correct execution flow within the JVM environment.

4.1.4 Virtual Machine

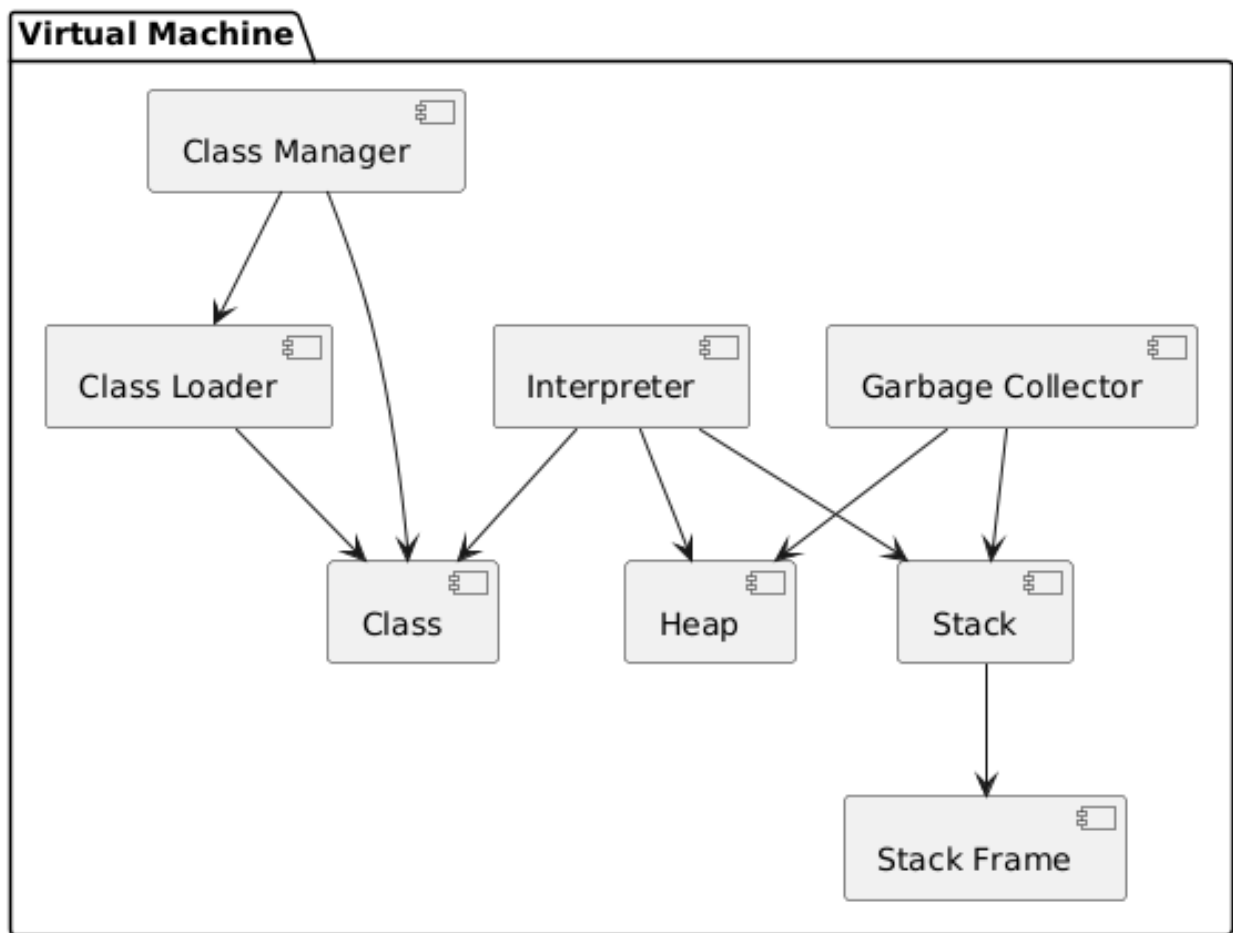


Figure 4.2: Component diagram of virtual machine

The Virtual Machine (VM) is the heart of the JVM’s architecture. It serves as the exe-

cution 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, heap 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.

- **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.

LoadedClass
Class Name: String
Super Class: Reference
Interfaces: References
Instance Fields: Field Information
Instance Fields Indices: HashMap<String, index>
Static Fields: FieldInfo
Static Field Indices: HashMap<String, use>
Static Values: Value[]
Methods: Methods Infos
Constant Pool: Reference
Access Flags: Class Access Flag
Code Cache: HashMap<NameDes, Code>>
Init State: Mutex<InitState>

Figure 4.3: Loaded class structure

- **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 and intermediate results. Every time a method is invoked, a new Stack Frame is created, which is pushed into the stack. This handles the parameters passed to the method. 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.

Frame
Constant Pool: Reference
Method Name Descriptor: NameDes
Code: Reference Code block
PC: 32-bit
Locals: Value[]
Operands: Stack of values

Figure 4.4: Frame structure

- 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.
- Heap:** The heap is responsible for dynamic memory allocation, where objects and arrays are stored during program execution. It is divided into generations to optimize garbage collection (GC). When an object or array is created, memory is allocated from the heap. The heap dynamically grows or shrinks based on memory requests. Newly created objects are typically allocated in the Young Generation, and as they survive multiple GC cycles, they are promoted to the Old Generation. This generational approach allows the JVM to perform frequent, low-cost collections in the Young Generation, while minimizing the impact on long-lived objects in the Old Generation. The heap ensures efficient memory usage by managing the allocation and deallocation of objects, while minimizing fragmentation and maximizing the performance of garbage collection.
- 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 UML Diagrams

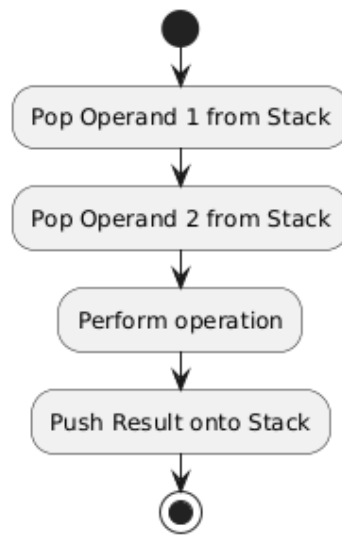


Figure 4.5: State diagram for arithmetic instructions

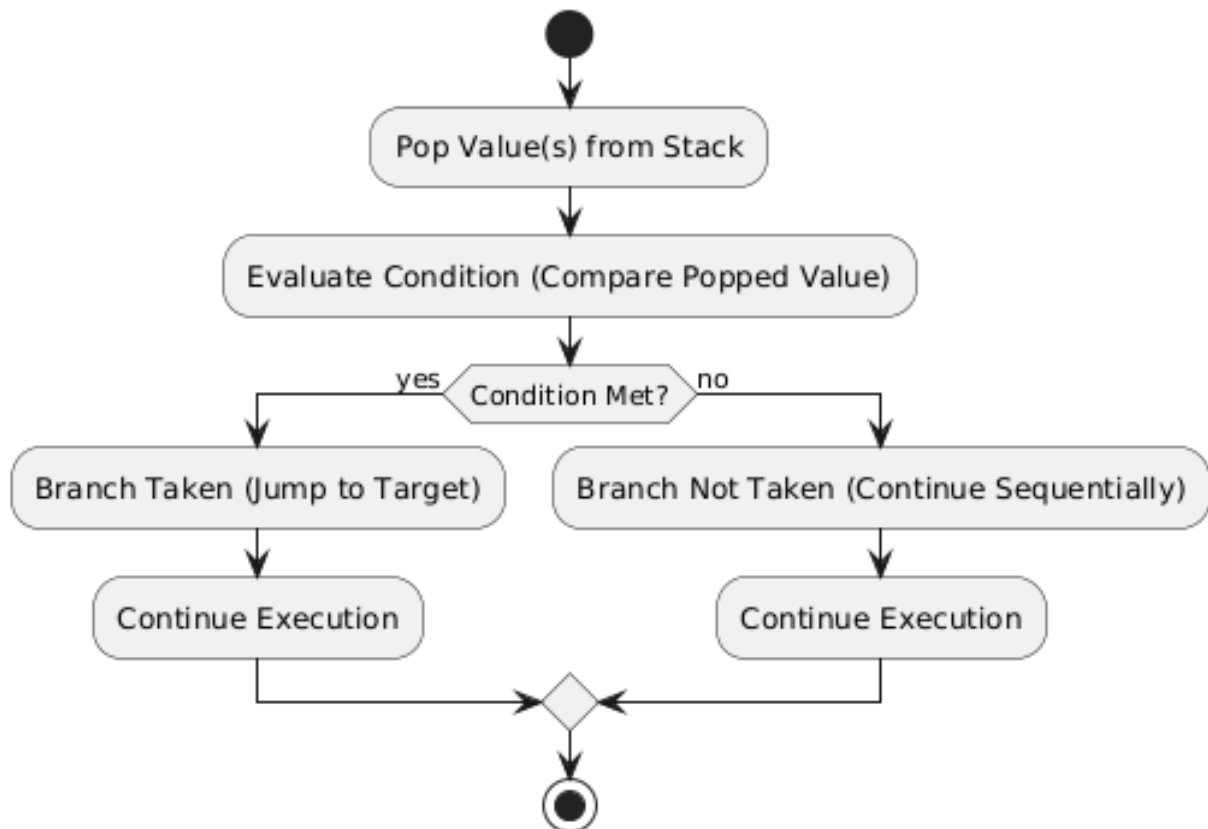


Figure 4.6: State diagram for branching instructions

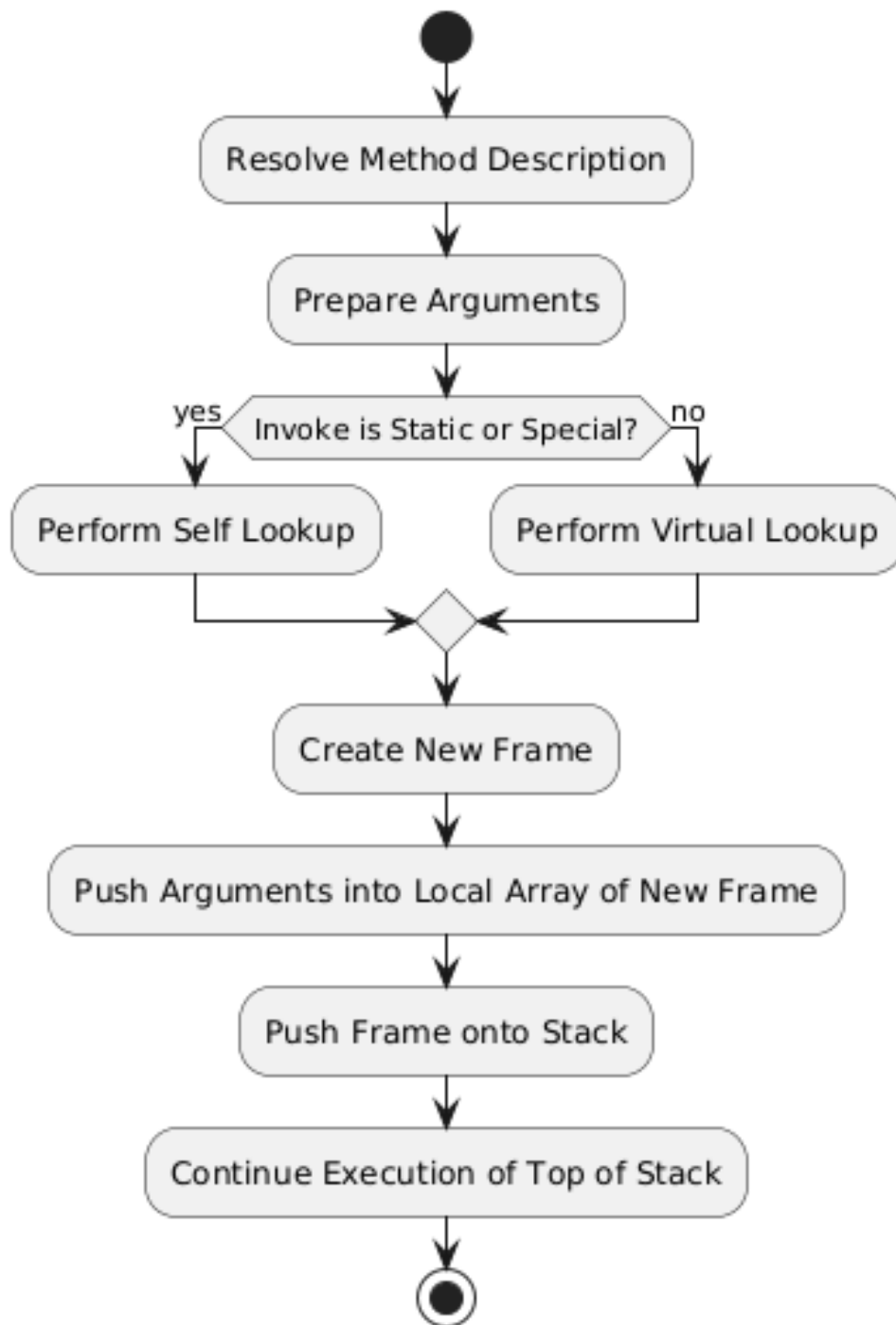


Figure 4.7: State diagram for method invocation

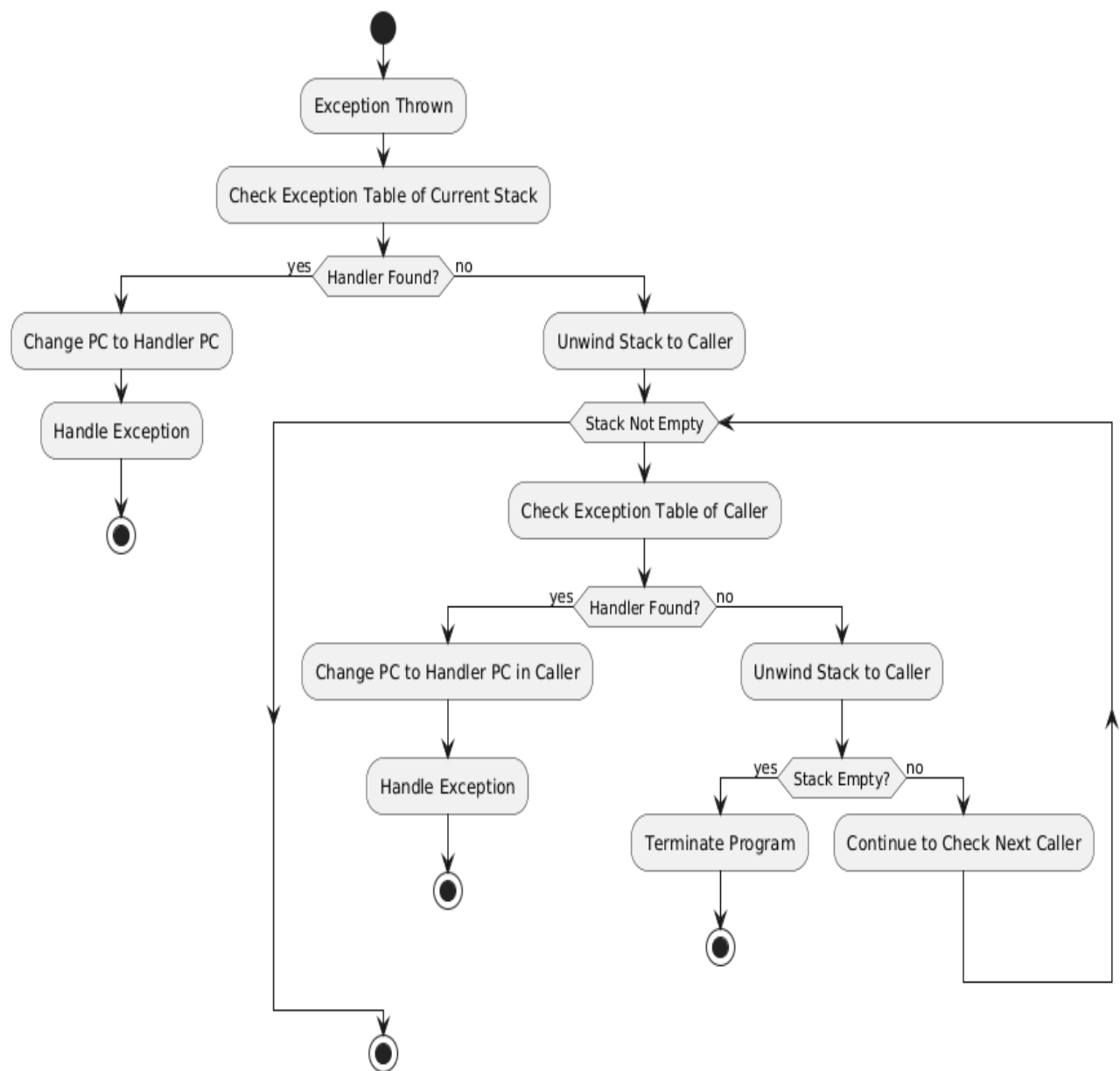


Figure 4.8: State diagram for exception handling

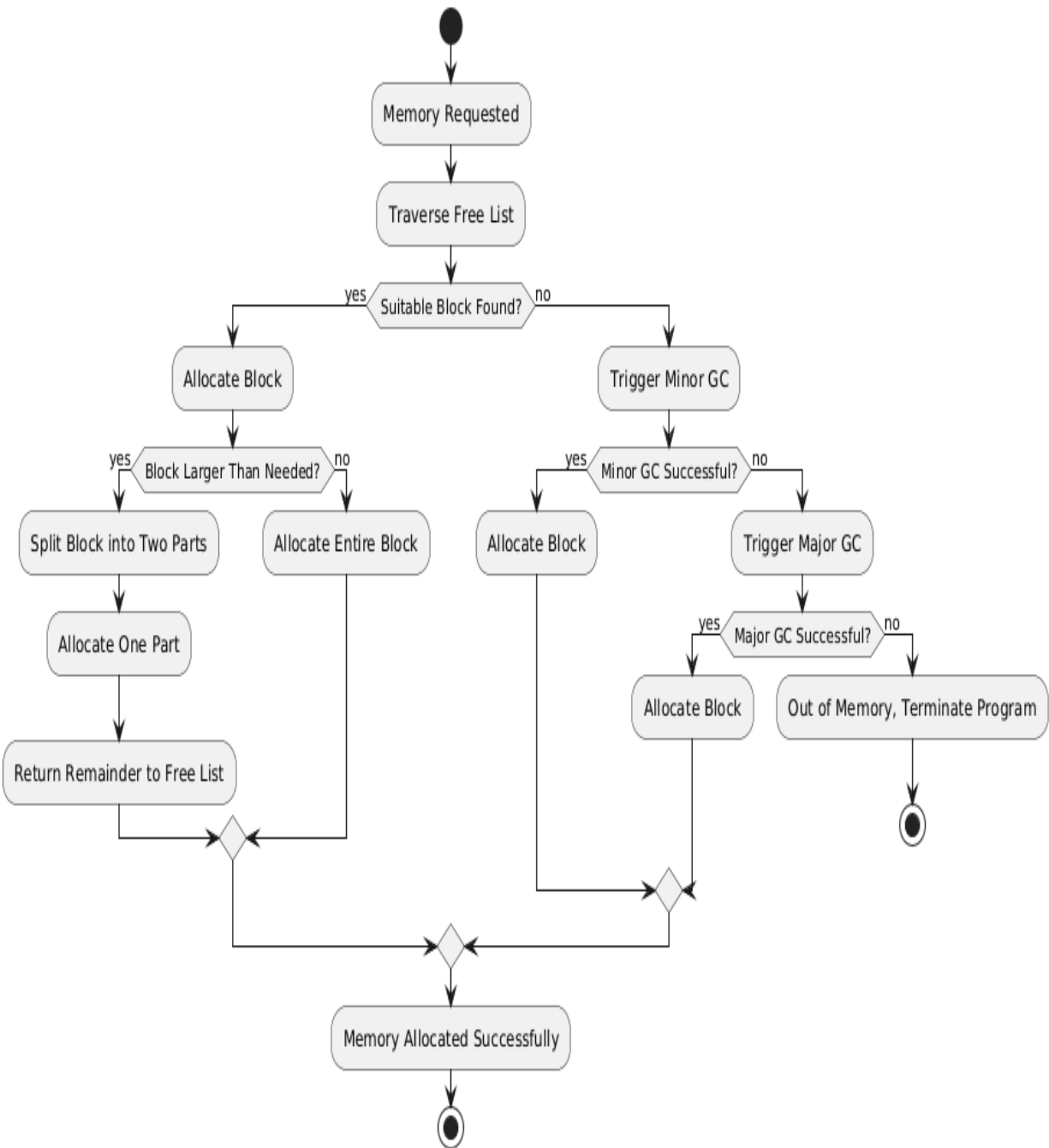


Figure 4.9: State diagram for memory allocation

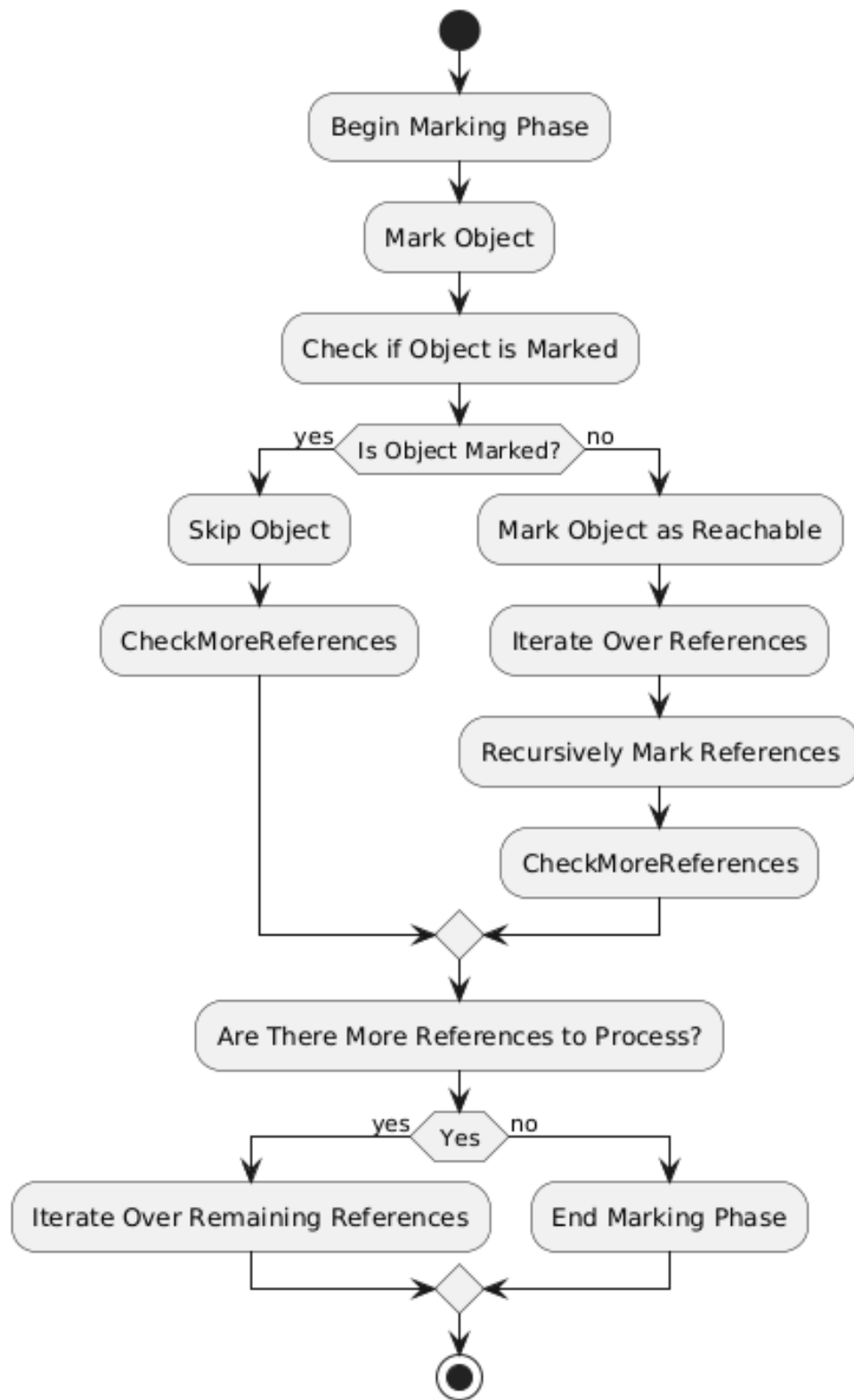


Figure 4.10: State diagram for mark object

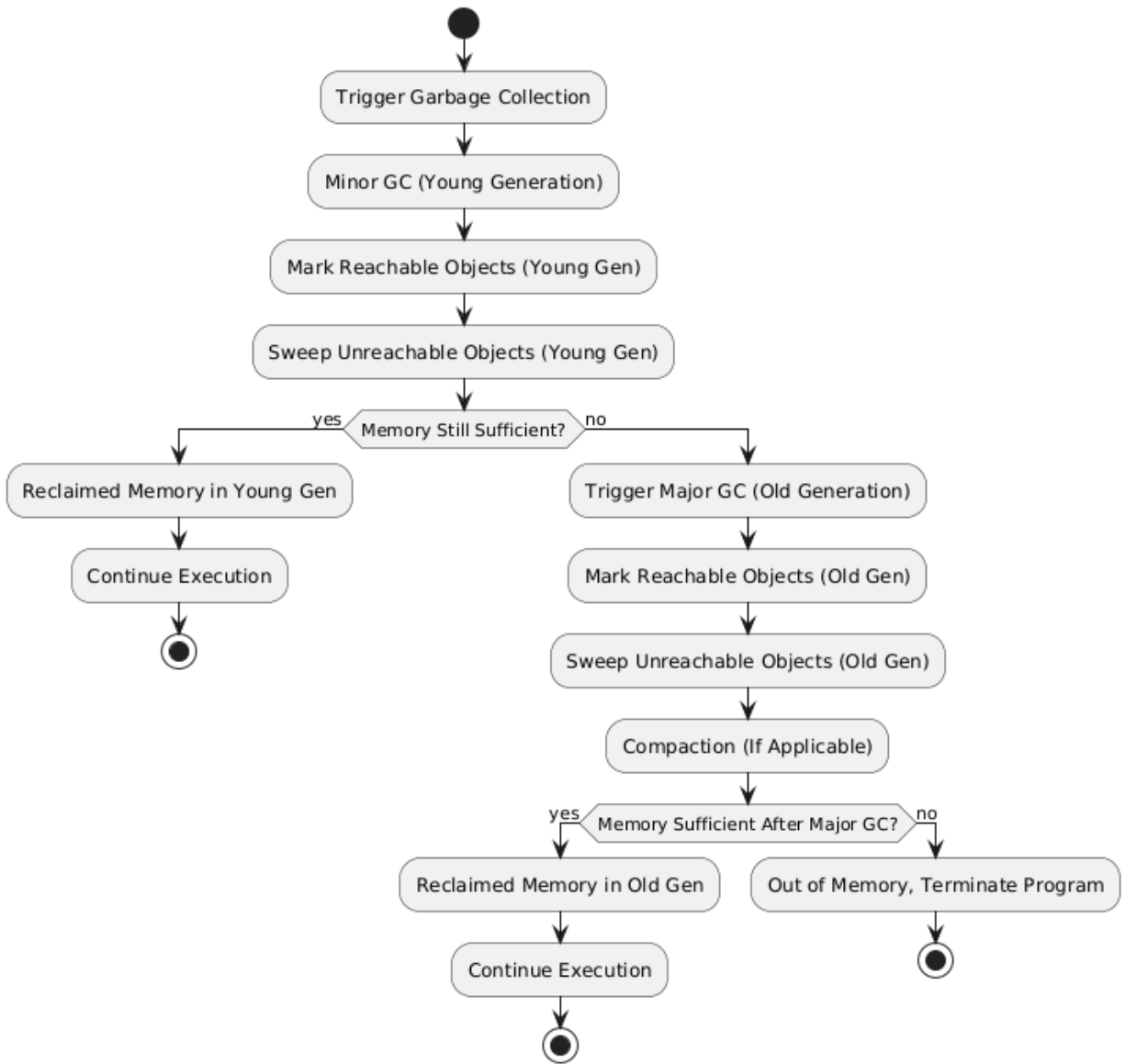


Figure 4.11: State diagram for garbage collection

4.3 Language and Tools

4.3.1 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.

4.3.2 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. Result and Discussion

5.1 Normal Arithmetic and branching program

Code

```
1 public class SumOfEvenNumbers {  
2     public static void main(String[] args) {  
3         int start = 1;  
4         int end = 100;  
5         int sum = 0;  
6  
7         for (int i = start; i <= end; i++) {  
8             if (i % 2 == 0) {  
9                 sum += i;  
10            }  
11        }  
12        ioTer.prints("Sum");  
13        ioTer.println(sum);  
14    }  
15 }
```

Output



```
Sum  
2550.000000
```

Figure 5.1: Output of program to calculate the sum of even numbers upto 100

5.2 Inheritance program

Code

```
1 // Base class
2 class Animal {
3     // Instance variables
4     String name;
5     int age;
6     // Constructor for Animal
7     public Animal(String name, int age) {
8         this.name = name;
9         this.age = age;
10    }
11    // Method to display animal information
12    public void speak() {
13        System.out.println("Animal is making a sound");
14    }
15    // Method to display general information about the animal
16    public void displayInfo() {
17        System.out.println(name);
18        System.out.println("Age: " + age);
19    }
20 }
21 // Subclass Dog inheriting from Animal
22 class Dog extends Animal {
23     // Constructor for Dog
24     public Dog(String name, int age) {
25         super(name, age); // Calls the constructor of the parent class (Animal)
26     }
27     // Overriding the speak method
28     @Override
29     public void speak() {
30         ioTer.prints("The dog barks");
31     }
32 }
33 // Subclass Cat inheriting from Animal
34 class Cat extends Animal {
```

```

35 // Constructor for Cat
36 public Cat(String name, int age) {
37     super(name, age); // Calls the constructor of the parent class (Animal)
38 }
39 // Overriding the speak method
40 @Override
41 public void speak() {
42     ioTer.prints("The cat meows");
43 }
44 }
45 // Main class to test inheritance
46 public class Main {
47     public static void main(String[] args) {
48         // Creating objects of Dog and Cat
49         Dog dog = new Dog("Buddy", 3);
50         Cat cat = new Cat("Whiskers", 2);
51
52         dog.displayInfo();
53         dog.speak();
54
55         cat.displayInfo();
56         cat.speak();
57     }
58 }

```

Output

```

Buddy
Age:
3.000000
The dog barks
Whiskers
Age:
2.000000
The cat meows

```

Figure 5.2: Output of program showcasing the inheritance

5.3 Interface program

Code

```
1 // Define the Shape interface
2 interface Shape {
3     double perimeter(); // Method to calculate the perimeter
4     double area();      // Method to calculate the area
5 }
6 // Rectangle class implements Shape interface
7 class Rectangle implements Shape {
8     double length;
9     double width;
10    // Constructor for Rectangle
11    public Rectangle(double length, double width) {
12        this.length = length;
13        this.width = width;
14    }
15    // Implement the perimeter method for Rectangle
16    @Override
17    public double perimeter() {
18        return 2 * (length + width);
19    }
20    // Implement the area method for Rectangle
21    @Override
22    public double area() {
23        return length * width;
24    }
25 }
26 class Triangle implements Shape {
27     double side1;
28     double side2;
29     double side3;
30     double base;
31     double height;
32
33     public Triangle(double side1, double side2, double side3, double base, double
        height) {
```

```

34     this.side1 = side1;
35     this.side2 = side2;
36     this.side3 = side3;
37     this.base = base;
38     this.height = height;
39 }
40 @Override
41 public double perimeter() {
42     return side1 + side2 + side3;
43 }
44 @Override
45 public double area() {
46     return 0.5 * base * height;
47 }
48 }
49 // Main class to test the interface and its implementations
50 public class Main {
51     public static void main(String[] args) {
52         // Create a Rectangle object
53         Rectangle rectangle = new Rectangle(5.0, 3.0);
54         ioTer.prints("Rectangle perimeter:");
55         ioTer.println(rectangle.perimeter());
56         ioTer.prints("Rectangle area:");
57         ioTer.println(rectangle.area());
58         // Create a Triangle object
59         Triangle triangle = new Triangle(3.0, 4.0, 5.0, 4.0, 3.0);
60         ioTer.prints("Triangle perimeter:");
61         ioTer.println(triangle.perimeter());
62         ioTer.prints("Triangle area:");
63         ioTer.println(triangle.area());
64     }
65 }

```

Output

```
Rectangle perimeter:
16.000000
Rectangle area:
15.000000
Triangle perimeter:
12.000000
Triangle area:
6.000000
```

Figure 5.3: Output of program showcasing the interface

5.4 Polymorphism program

Code

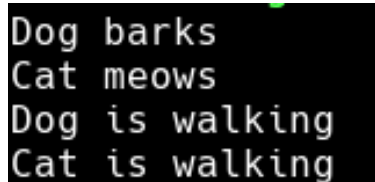
```
1 // Defining an interface with abstract methods
2 interface Animal {
3     void sound(); // Abstract method to be implemented by classes
4     void move(); // Abstract method to be implemented by classes
5 }
6 // Dog class implements the Animal interface
7 class Dog implements Animal {
8     // Implementing the sound method (runtime polymorphism)
9     @Override
10    public void sound() {
11        ioTer.prints("Dog barks");
12    }
13
14    // Implementing the move method (overloading and runtime polymorphism)
15    @Override
16    public void move() {
17        ioTer.prints("Dog is walking");
18    }
19
20 }
21 // Cat class implements the Animal interface
22 class Cat implements Animal {
23     // Implementing the sound method (runtime polymorphism)
24     @Override
```

```

25     public void sound() {
26         ioTer.prints("Cat meows");
27     }
28
29     // Implementing the move method (overloading and runtime polymorphism)
30     @Override
31     public void move() {
32         ioTer.prints("Cat is walking");
33     }
34 }
35 public class Main {
36     public static void main(String[] args) {
37         // Creating objects of Dog and Cat using the Animal interface
38         Animal myDog = new Dog();
39         Animal myCat = new Cat();
40
41         // Demonstrating runtime polymorphism (method overriding)
42         myDog.sound(); // Calls Dog's sound() method
43         myCat.sound(); // Calls Cat's sound() method
44
45         // Demonstrating method overloading (compile-time polymorphism)
46         myDog.move(); // Calls the Dog's move() method
47         myCat.move();
48         //myDog.move("jumping"); // Calls the Dog's move() method with parameter
49     }
50 }

```

Output



```

Dog barks
Cat meows
Dog is walking
Cat is walking

```

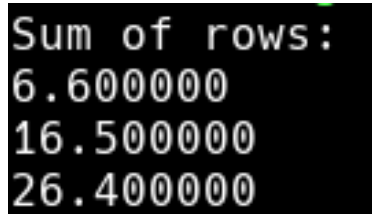
Figure 5.4: Output of program showcasing the polymorphism

5.5 Multidimensional array program

Code

```
1 public class MultiDimensionalArrayExample {
2     public static void main(String[] args) {
3         // Creating a 2D array of doubles (3x3 for example)
4         double[][] matrix = {
5             {1.1, 2.2, 3.3},
6             {4.4, 5.5, 6.6},
7             {7.7, 8.8, 9.9}
8         };
9         // Creating a 1D array of doubles to store the sum of each row
10        double[] rowSums = new double[matrix.length];
11
12        // Loop through each row of the 2D array and calculate the sum
13        for (int i = 0; i < matrix.length; i++) {
14            double sum = 0; // Variable to store the sum of the current row
15            for (int j = 0; j < matrix[i].length; j++) {
16                sum += matrix[i][j]; // Add each element of the row to the sum
17            }
18            rowSums[i] = sum; // Store the sum of the row in the 1D array
19        }
20        ioTer.prints("Sum of rows:");
21        for (int i = 0; i < rowSums.length; i++){
22            ioTer.println(rowSums[i]);
23        }
24    }
25 }
```

Output



```
Sum of rows:
6.600000
16.500000
26.400000
```

Figure 5.5: Output of program showcasing the multidimensional array

5.6 Exception handling program

Code

```
1 public class ExceptionHandlingExample {
2     public static void main(String[] args) {
3         // Division by zero handling
4         try {
5             int result = divide(10, 0); // Attempt to divide by zero
6             ioTer.prints("Division Result:");
7             ioTer.println(result);
8         } catch (ArithmeticException e) {
9             ioTer.prints("Error: Division by zero is not allowed.");
10        } finally {
11            ioTer.prints("Finally block after division by zero.");
12        }
13        // Array index out of bounds handling
14        try {
15            int[] numbers = {1, 2, 3};
16            int five = numbers[5];
17            ioTer.prints("Accessing element at index 5: ");
18            ioTer.println(numbers[5]); // Attempt to access an invalid index
19        } catch (ArrayIndexOutOfBoundsException e) {
20            ioTer.prints("Error: Array index out of bounds.");
21        } finally {
22            ioTer.prints("Finally block after array index out of bounds.");
23        }
24    }
25    // Method to perform division
```

```
26     public static int divide(int a, int b) {  
27         return a / b; // Division operation that may cause ArithmeticException  
28     }  
29 }
```

Output

```
Error: Division by zero is not allowed.  
Finally block after division by zero.  
Error: Array index out of bounds.  
Finally block after array index out of bounds.
```

Figure 5.6: Output of program showcasing the exception handling

6. Conclusion

This project has successfully designed and implemented a lightweight Java Virtual Machine (JVM) capable of executing a subset of the Java programming language. By focusing on core functionalities such as class loading, bytecode interpretation, memory management, and garbage collection, we have demonstrated the feasibility of a simplified JVM that balances applicability and educational value.

Through this implementation, we have gained a deeper understanding of virtual machine architecture, bytecode execution, and runtime memory management. This project serves as both a practical tool and an academic resource, offering a simplified look into the intricacies of JVMs without the complexity of full-scale implementations. This work lays a foundation for further enhancements, including Just-In-Time (JIT) compilation, advanced garbage collection strategies, and runtime monitoring tools, contributing to the broader study of virtual machines and programming language execution environment.

7. Limitations and Future Enhancements

7.1 Limitations

- The JVM does not support Just-In-Time (JIT) compilation, leading to reduced execution speed compared to optimized JVMs.
- The current prototype doesn't support multithreading features for execution of Java programs concurrently.
- Input/output (I/O) handling, including file operations, network communication, and asynchronous I/O, is not supported other than basic terminal output.
- The *invokedynamic* instruction is not implemented, limiting support for lambda functions and dynamically resolved method calls.
- Full compatibility with Java's standard libraries is not provided, restricting the execution of programs that depend on extensive built-in APIs.

7.2 Future Enhancements

- **Just-In-Time Compilation** – JIT compilation support would enhance performance by compiling frequently executed bytecode into machine code dynamically during runtime, bypassing interpretation overhead and speeding execution. This would maintain the JVM's efficiency without sacrificing platform independence.
- **Command Line Options for Runtime Configuration** – The introduction of configurable CLI parameters would allow runtime variables such as memory sizes, garbage collection policies, and debugging flags to be adjusted. This would increase usability as developers could better match JVM behavior with their application and desired level of performance.
- **Bytecode Verification** – Incorporating a bytecode verifier would add more security and stability as it would ensure loaded bytecode conforms to JVM constraints to prevent possible stack underflow, illegal memory access, or type mismatch problems. The process would play a vital role in executing untrusted or third-party compiled code securely.

- **GUI for Monitoring** – A graphical user interface for monitoring JVM runtime would provide real-time data regarding memory usage, threads currently running, method runtime, and garbage collection events. This would help in debugging, performance tuning, and understanding the internal activities of the JVM through visual representation.

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.
- [15] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *Memory Management in the Java HotSpot™ Virtual Machine*, 2006. Java SE 22 Edition.