TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A

PROJECT PROPOSAL

ON

A JAVA VIRTUAL MACHINE FOR SUBSET OF JAVA

SUBMITTED BY:

LOKESH PANDEY (PUL077BCT040)

MANDIP THAPA (PUL077BCT044)

MANISH KUNWAR (077BCT045)

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING

May, 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), IOE.

Our sincere thanks to Department of Electronics and Computer Engineering(DoECE) for providing us the inspiration and guidance for this project.

We extend our gratitude to our cluster leader, Assistant Professor Dr. Basanta Joshi for helping us in idea brainstorming and initial draft.

We would also like to express our deep appreciation and gratitude to our friends for providing feedback and suggestions while selecting the project topic.

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 that runs programs written in Java programming language as well as other languages that are compiled to Java bytecode. It provides platform-independent runtime environment to run computer programs and supports Java's philosophy of "write once run everywhere" philosophy. Features of JVM includes automatic memory management using garbage collection, dynamic loading of classes. It abstracts the hardware level machine instruction allowing programmers to focus on application development, and provides freedom from platform-specific concerns.

This project aims to develop a Java Virtual Machine (JVM) for execute a subset of Java programming language. The proposed JVM will be able to parse and execute .class files obtained from compiling .java files. Key functionalities provided by the proposed system are exception handling, stack tracing and garbage collection. The Java language features planned to be supported are primitive types, arrays, strings, control flow statements, classes, sub-classes, interfaces, and methods(virtual and static). Some language features excluded are reflection, multithreading and just in time execution (JIT).

1.2 Problem statements

Understanding existing JVM implementations is a complex and daunting task because of their extensive feature sets and intricate internal workings. By concentrating on a subset of Java language features, this project aims to elucidate the processes involved in interpreting bytecode and executing it. This focused approach seeks to address the difficulties in understanding and analyzing the functionalities of current JVM. Therefore, this project is undertaken to systematically explore the internal workings of JVM.

1.3 Objectives

The primary objective of the project is to develop a Java Virtual Machine (JVM) that is able to execute the bytecode for a subset of features of java programming language. Another objective of the project is to gain an deeper understanding of the working of virtual machines in general and the engineering behind the implementation of its components and design choices.

1.4 Scope

Since the project promises to develop a Java Virtual Machine, it can be used as an entry point for learning about virtual machines for future explorers. Beyond practical application, this project aims to advance our understanding of virtual machines and program execution in the academic context.

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 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.[4] The most recent

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

Jikes Research Virtual Machine (Jikes RVM), 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, 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 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.

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 manipula-

tion, 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.

2.2.3 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. 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.

2.2.4 Garbage Collection

Garbage collection (GC) is a type of autonomous memory management used in computer science. 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.

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. Proposed Methodology

3.1 Feasibility Study

This feasibility study explores the prospects of developing a "Java Virtual Machine for 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 project is technically feasible due to its matured technology, numerous open-source implementations, vast developer pool, and extensive development tools.

3.1.2 Economic Feasibility

Our own personal computers are to be used to implement and examine the intended outcome of the project, resulting in almost no financial outlay. This shows that the project is economically feasible.

3.1.3 Time Feasibility

The time frame of nearly 10 months is presented to us for the completion of the project which is plenty. The timeline section[chapter 5] includes a thorough chronology that justifies the time feasibility.

3.2 Requirement Analysis

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.

4. Proposed System design

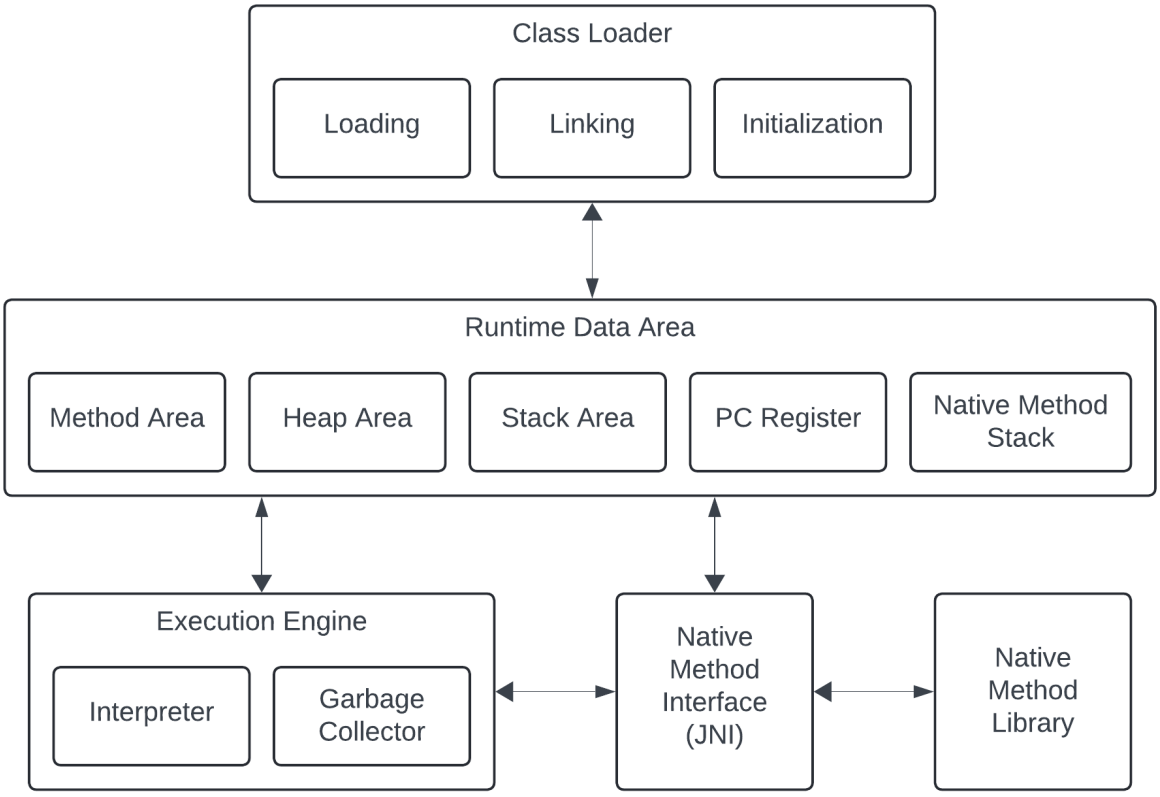

Figure 4.1: Proposed architecture

4.1 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.

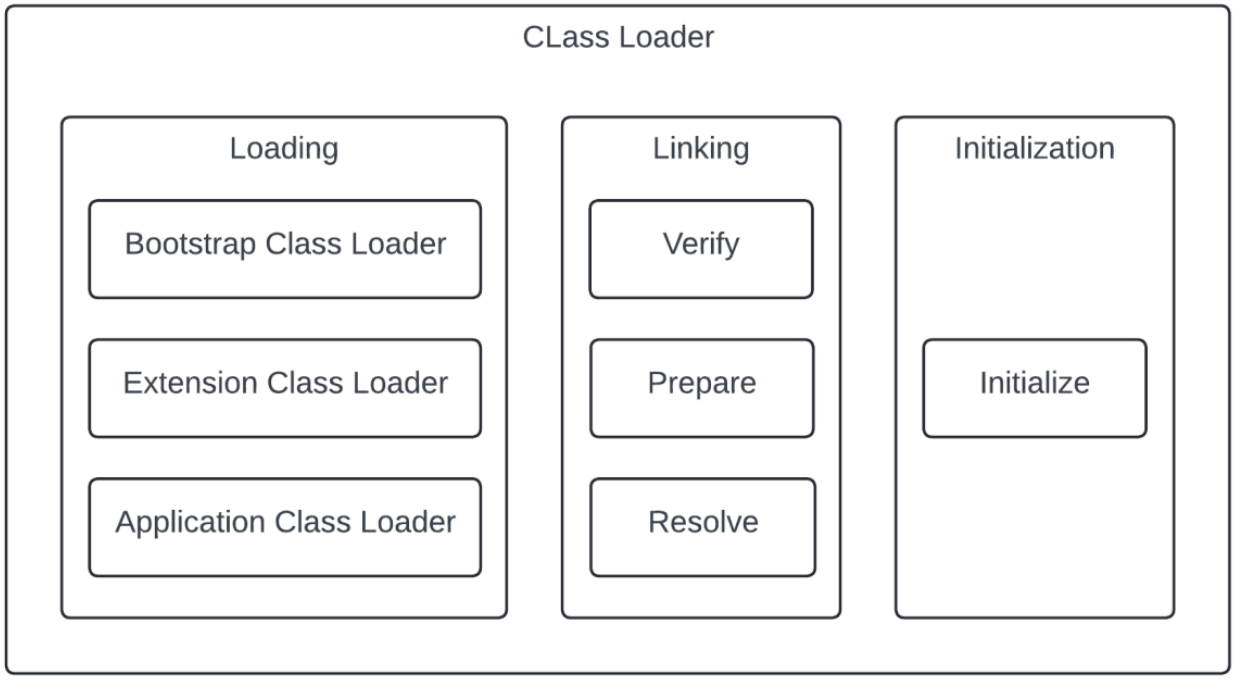

Figure 4.2: Class loader

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.2 Runtime data area

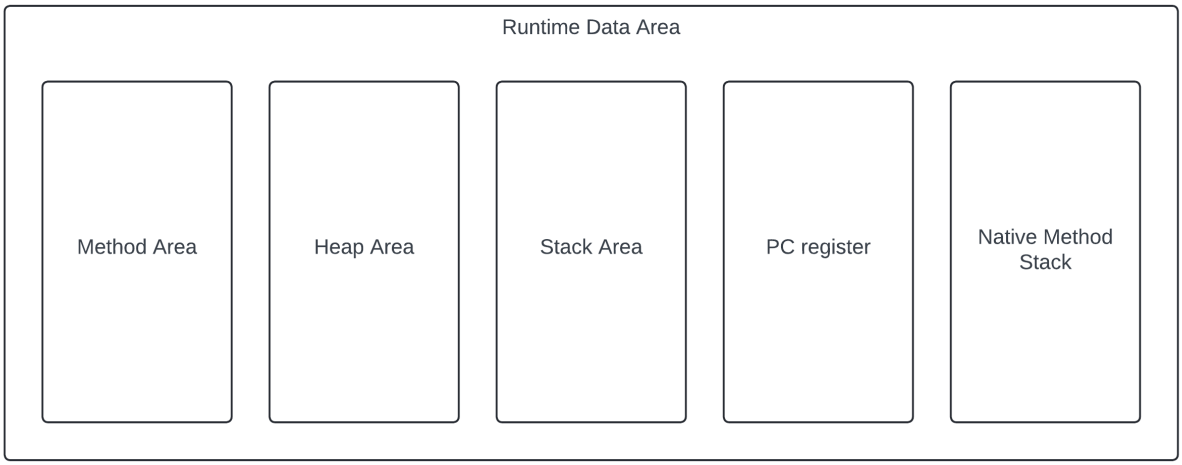

Figure 4.3: Runtime data area

4.2.1 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.

4.2.2 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.

4.2.3 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 vari-

ables. 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.

4.2.4 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.

4.2.5 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++.

4.3 Execution engine

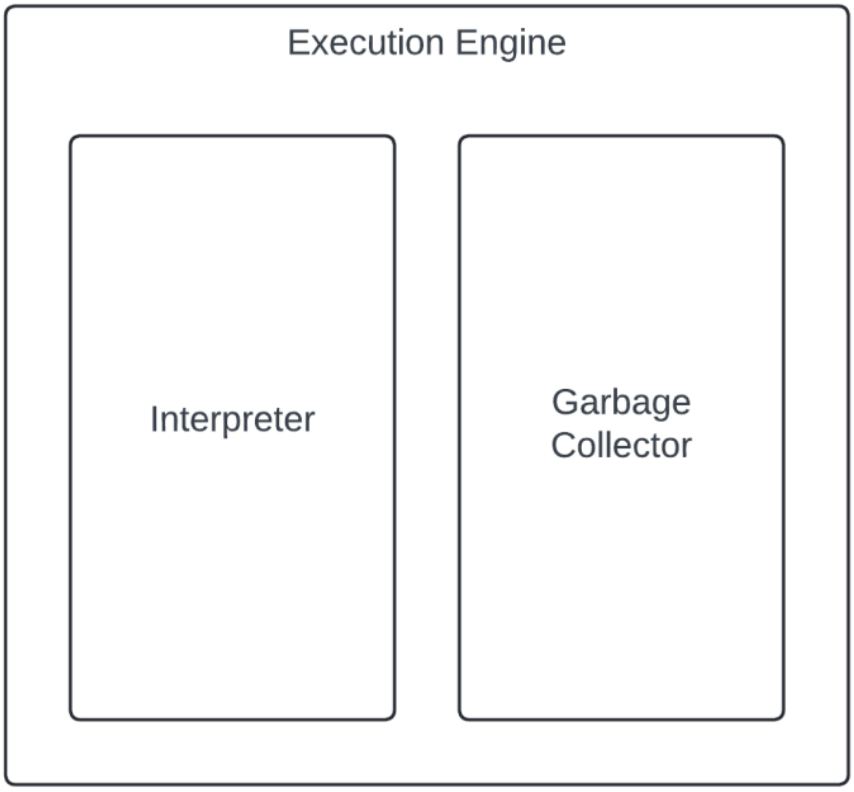

Figure 4.4: Execution engine

4.3.1 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.

4.3.2 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.

4.4 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.

4.5 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.

5. Timeline

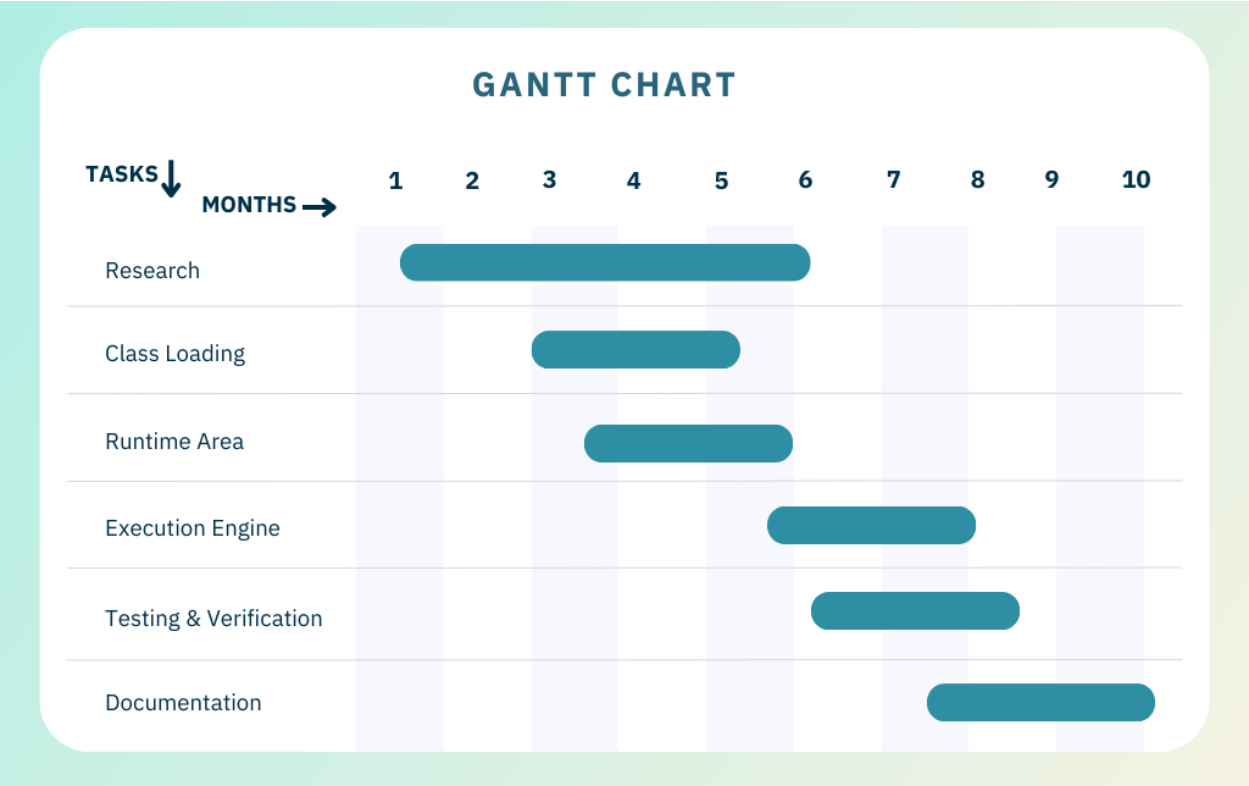

Figure 5.1: Timeline (Gantt Chart Form)

References

[1] Chen Fishbein and Shai Almong. What is codename one. *codenameone.blogpost.com*, 2012.

[2] Leonardo Zanivan. New open source jvm optimized for cloud and microservices. *medium.com*, 2018.

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

[4] Scott Lynn. For building programs that run faster anywhere: Oracle graalvm enterprise edition. *blogs.oracle.com*, 2019.