**COMPILER CONSTRUCTION LAB**

BSCS-[5A]

**LAB RESEARCH BASE DOCUMENT**

A picture containing text, emblem, symbol, crown

Description automatically generated

**Submitted by:**

**ROHAAN NADEEM (03-134202-102)**

**HAMZA KHALID (03-134182-021)**

**ABDUL QADEER (03-134202-034)**

**ub   Submitted to: Ms. Sadia Aslam**

**Submitted To:**

**MISS NIDA HAFEEZ**

**RESEARCH BASE PROJECT**

**Title:-** **Front end of Java Compiler**

**DEPARTMENT OF COMPUTER SCIENCES**

**BAHRIA UNIVERSITY, LAHORE CAMPUS**

**Front end of Java Compiler**

**Rohaan Nadeem#1, Hamza Khalid#2, Abdul Qadeer#3**

***BS Computer Science, Bahria University Lahore Campus***

***Johar Town, Lahore, Pakistan***

1 [rohaannadeem2@gmail.com](mailto:rohaannadeem2@gmail.com)

2 [hamzay9212@gmail.com](mailto:hamzay9212@gmail.com)

3 [sheikhqadeer83@gmail.com](mailto:sheikhqadeer83@gmail.com)

1. **INTRODUCTION:**

The front end of the Java compiler is a crucial component responsible for processing and analyzing Java source code, facilitating the transformation of human-readable code into executable bytecode. Since its inception, the front end of the Java compiler has undergone significant evolution to keep pace with the advancements in the Java programming language and meet the demands of an ever-changing software development landscape. This research paper aims to provide a comprehensive overview of the historical progression of the front end of the Java compiler, analyze its current state, and explore potential advancements for the future.

In its early stages, the front end of the Java compiler focused primarily on ensuring syntactic correctness and detecting simple semantic errors. Manual recursive descent parsers were used to parse the source code and generate an abstract syntax tree (AST). However, as the Java language evolved, introducing new language features and syntax enhancements, the front end had to adapt to handle these complexities. The need for more advanced parsing techniques and semantic analysis capabilities arose to support features like generics, lambdas, and modules.

Over the years, advancements in parsing techniques have greatly influenced the front end of the Java compiler. Parser generators such as ANTLR, JavaCC, and JFlex emerged, providing automatic generation of parsers from grammar specifications. These tools alleviated the burden of manual parser construction, enabling more efficient and flexible parsing approaches. Additionally, parser combinators gained popularity as a concise and modular method of building parsers using functional programming principles, offering improved readability and flexibility in handling complex language structures.

Semantic analysis within the front end has also seen notable improvements. Type checking algorithms have become more sophisticated, capable of handling complex type systems and enforcing stricter type rules. Symbol table construction and management have been optimized, facilitating faster and more accurate identifier resolution. Control flow analysis techniques have been refined, enabling better optimization opportunities and enhanced static analysis capabilities.

Looking towards the future, the front end of the Java compiler holds great potential for further advancements. As the Java language continues to evolve, the front end must adapt to support new language features and syntax enhancements. Moreover, with the increasing emphasis on performance and efficiency in software development, future advancements in the front end could focus on optimizing the parsing process, improving error handling and recovery mechanisms, enhancing semantic analysis algorithms, and exploring novel approaches for static analysis and program understanding.

In conclusion, the front end of the Java compiler has undergone significant evolution to support the advancements in the Java language and meet the evolving needs of software development. The advancements in parsing techniques, semantic analysis, and tooling have played a vital role in maintaining the front end's efficiency and effectiveness. The future of the front end holds exciting possibilities for further enhancements, with a focus on optimizing parsing, improving error handling, and exploring novel approaches driven by advancements in language features and emerging technologies. By continually advancing the front end of the Java compiler, developers can benefit from improved productivity, code quality, and performance in Java software development.

1. **INITIAL PHASE OF JAVA COMPILER CONSTRUCTION**

Programmers typically write language statements in a given programming language one line at a time using a code editor or an integrated development environment (IDE). The resulting file contains what are called the source statements. The programmer then runs a compiler for the appropriate language, specifying the name of the file that contains the source statements.

1. *Introduction of JAVA:*

The Java TM programming language is designed to meet the challenges of application development in the context of heterogeneous, network-wide distributed environments. Paramount among these challenges is secure delivery of applications that consume the minimum of system resources, can run on any hardware and software platform, and can be extended dynamically.[1]

The Java programming language originated as part of a research project to develop advanced software for a wide variety of network devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating platform. When the project started, C++ was the language of choice. But over time the difficulties encountered with C++ grew to the point where the problems could best be addressed by creating an entirely new language platform. Design and architecture decisions drew from a variety of languages such as Eiffel, Small Talk, Objective C, and Cedar/Mesa. The result is a language platform that has proven ideal for developing secure, distributed, network-based end-user applications in environments ranging from network-embedded devices to the World-Wide Web and the desktop.

A Java compiler is a program that takes the text file work of a developer and compiles it into a platform-independent Java file. Java compilers include the Java Programming Language Compiler (javac), the GNU Compiler for Java (GCJ), the Eclipse Compiler for Java (ECJ) and Jikes.

1. *Inventor of Java*

**James Gosling** [3] his pic given in fig 1 is widely credited as the inventor of the Java programming language. He developed Java while working at Sun Microsystems in the **mid-1990s**. Gosling and his team created Java with the goal of developing a platform-independent language that could be used to build applications that would run on any system. Java quickly gained popularity and became one of the most widely used programming languages for a variety of applications, including web development, mobile app development, and enterprise software.



fig 1 James Gosling

1. *Develop Java compiler front end for the first time:*

**Arthur van Hoff**.[4] his pic given in fig 2. He was a member of the original team that developed Java at **Sun Microsystems**. Van Hoff worked on the design and implementation of the Java programming language, including the creation of the compiler front end. The initial work on the Java compiler began in the early **1990s**, and the first version of the Java programming language was **released in** **1995**. Van Hoff's contributions were instrumental in shaping the language and making it accessible to developers.



Fig 2 **Arthur van Hoff**

1. *Evolution of the Java Language Specification:*

The Java front end, which is responsible for parsing and analyzing Java source code, has evolved over the years to improve performance, language support, and error detection. Here's a brief overview of the evolution of the Java front end and eg [5] TABLE 1:

=> Original Java Compiler (JDK 1.0): The first version of the Java front end was part of the original Java Development Kit (JDK) 1.0 release in 1995. It provided basic parsing and type-checking capabilities.

Java Compiler API (JDK 1.2): With the release of JDK 1.2 in 1998, the Java Compiler API (javax.tools) was introduced. It allowed developers to programmatically access and invoke the Java compiler. This provided more flexibility in integrating the Java compiler into development tools and IDEs.

=> Java Compiler Compiler (JavaCC): JavaCC is a parser generator used to generate the Java front end's parser code. It became popular as a tool for creating language-specific front ends, including Java. JavaCC simplifies the process of creating parsers by using a grammar-based approach.

=> Java Parser API (Java 6): Java 6 introduced the Java Parser API (javax.lang.model), which provides a standardized way to access and analyze the structure of Java source code. It allows tools to perform tasks such as code generation, static analysis, and refactoring. This API made it easier for external tools and libraries to interact with the Java front end.

=> Java Compiler Tree API (Java 7): In Java 7, the Java Compiler Tree API (com.sun.source) was introduced. It provided a higher-level abstraction for accessing and manipulating the abstract syntax tree (AST) of Java source code. The AST represents the structure and semantics of the code, enabling advanced code analysis and transformation.

=> Project Amber: Project Amber, an ongoing OpenJDK project, focuses on enhancing the Java language with productivity-oriented features. Some of these enhancements include simplified syntax, pattern matching, and improved type inference. These changes impact the front end by extending the capabilities of the parser and type-checker to support new language constructs.

=> External Libraries: Alongside the evolution of the official Java front end, various external libraries and tools have been developed to work with Java source code. Examples include libraries like Eclipse JDT (Java Development Tools) and IntelliJ IDEA's internal Java front end, which provide advanced code analysis, refactoring, and error detection capabilities.

Overall, the evolution of the Java front end has aimed to improve the developer experience, increase language expressiveness, and provide better tooling support for code analysis and transformation.

1. *Design Goals and Principles:*

The design requirements of the Java TM programming language are driven by the nature of the computing environments in which software must be deployed.

The massive growth of the Internet and the World-Wide Web leads us to a completely new way of looking at development and distribution of software. To live in the world of electronic commerce and distribution, Java technology must enable the development of secure, high performance, and highly robust applications on multiple platforms in heterogeneous, distributed networks.

Operating on multiple platforms in heterogeneous networks invalidates the traditional schemes of binary distribution, release, upgrade, patch, and so on. To survive in this jungle, the Java programming language must be architecture neutral, portable, and dynamically adaptable.

The system that emerged to meet these needs is simple, so it can be easily programmed by most developers; familiar, so that current developers can easily learn the Java programming language; object oriented, to take advantage of modern software development methodologies and to fit into distributed client-server applications; multithreaded, for high performance in applications that need to perform multiple concurrent activities, such as multimedia; and interpreted, for maximum portability and dynamic capabilities.[2]

Together, the above requirements comprise quite a collection of buzzwords, so let's examine some of them and their respective benefits before going on.

1. *Challenges and Solutions:*

Java Front end challenges were sorted by time with the help of enhancement in the lexical analysis, syntactic analysis, semantic analysis, intermediate code, code optimization, and code creation are all aspects of the compiler.

To make an easy Front-end environment for developer to make application with guidelines with the help of these phases narrated above.

Table 1 [5]

Phases of Compiler

|  |
| --- |
| Source Program |

|  |
| --- |
| Lexical Analysis Phase |

Token

|  |
| --- |
| Syntax Analysis Phase |

Syntax Tree

|  |
| --- |
| Semantic Analysis Phase |

Parse Tree

|  |
| --- |
| Intermediate Code Phase |

Intermediate Code

|  |
| --- |
| Code Optimization |

Optimized Code

|  |
| --- |
| Code Generation |

|  |
| --- |
| Target Program |

1. *Impact and Legacy:*

The starting phase of Java compiler construction played a significant role in the adoption and success of the Java language. The design and implementation of the Java compiler were crucial in shaping the language's features, performance, and compatibility. Here are some key aspects to consider:

=> Language Design and Features:

During the construction of the Java compiler, the language designers focused on creating a language that was simple, object-oriented, and platform-independent. This design choice made Java more accessible to developers with different programming backgrounds and facilitated the transition to object-oriented programming.

=> Performance Optimization:

The efficiency of the Java compiler had a direct impact on the adoption of the language. Early versions of the Java compiler faced performance challenges, as the initial implementations relied heavily on interpretation.

1. **CURRENT PHASE OF JAVA COMPILER CONSTRUCTION**
2. *Introduction:*

There are not many technologies that can brag about staying relevant for more than 20 years. But this year, Java was voted the 5th most popular technology, eclipsed only by undisputed leaders JavaScript, HTML, CSS, and SQL. While it’s 18th on the list of most loved in the same StackOverflow survey, it’s also way down the list in its most-dreaded ranking. Today we unravel the many successes and challenges of Java – the time-honored technology with the iconic steaming cup-of-coffee logo fig 3, a language near and dear to many programmers today.

**

Fig 3 Java Logo

*=> Java as Frontend Development:*

When the browser matured and became a clear difference between backend and frontend programming models, Java shifted towards server-side work. Today, Java is rarely used in frontend development. Java is not JavaScript – these are two different programming languages.

Java is perfect for writing whole apps with complicated logic, large or complex data sets, and desktop-style interfaces.

1. *Achieved Goals:*

In the field of Java compiler construction, several goals have been achieved to date. These goals primarily focus on enhancing the performance, functionality, and usability of the Java compiler. Here are some of the key goals that have been accomplished:

*=>* Performance Optimization:

Significant advancements have been made in optimizing the performance of the Java compiler. Techniques such as Just-In-Time (JIT) compilation, adaptive optimization, and profile-guided optimization have been implemented to improve the runtime performance of Java applications. These optimizations help to dynamically compile frequently executed code segments to native machine code, resulting in faster execution and reduced memory footprint.

*=> Language Evolution:*

The Java compiler has played a vital role in supporting the evolution of the Java language. The introduction of new language features required enhancements in the compiler to parse, validate, and generate bytecode for these features. The Java compiler continues to evolve to support the latest language features and specifications.

*=> Language Tooling:*

The development of advanced Integrated Development Environments (IDEs) and other tooling has been a major focus in the field of Java compiler construction. So that they provide rich functionality, including intelligent code completion, syntax highlighting, refactoring, debugging, and code analysis.

*=> Error Detection and Diagnostics:*

The Java compiler has become more sophisticated in terms of detecting errors and providing useful diagnostics to developers. Compiler messages and warnings have been improved to provide better insights into coding mistakes, potential performance issues, and compatibility problems. This helps developers identify and fix issues early in the development process, improving code quality and maintainability.

*=> Compatibility and Interoperability:*

The Java compiler ensures compatibility and interoperability across different versions of Java and various Java Virtual Machine (JVM) implementations. It enforces the Java Language Specification (JLS) to maintain consistency and predictability in how Java code is compiled and executed.

1. Language Enhancements:

After examination of the recent language features introduced in Java compiler the front end also improved in the phases of compiler construction, many versions of Java Languages are released whose example is in Table 2 or in the eg. [6], which makes code more efficient and a programmer friendly environment to code easily.

1. Build Systems and Compilation Efficiency:

With the help of advancement in the version of Java Compiler and its language, Java is more helpful to programmers to develop applications, and building systems efficiently and due to enhancement in compiler front end, compilation it helps developers to work faster, easily, and efficiently.

1. Case Studies and Research Projects:

According to the research and data we got related to the current phase of Java compiler construction, we can provide you with an analysis of some notable research projects and case studies that have contributed to the advancement of Java compiler construction in the past. Here are some examples which may not reflect the most recent developments in the field:

=> GraalVM:

GraalVM eg fig 4 is an open-source project that aims to provide a high-performance runtime for Java applications. It includes a Java compiler called Graal Compiler, which utilizes just-in-time (JIT) compilation techniques and offers significant performance improvements over traditional JVM compilers.

A black background with blue and orange letters

Description automatically generated with low confidence

Fig 4 GraalVM Logo

=> OpenJDK:

OpenJDK eg fig 5 is an open-source implementation of the Java Development Kit (JDK). The project involves the collaborative efforts of developers worldwide to improve and enhance the Java platform. The Java compiler component within OpenJDK undergoes continuous development and improvement, with a focus on language features, performance optimizations, and compatibility.



Fig 5 OpenJDK

=> Tooling and Developer Support:

The construction of the Java compiler resulted in the development of robust tooling and developer support ecosystems. Integrated Development Environments (IDEs) like Eclipse, IntelliJ IDEA, and NetBeans, along with build tools like Maven and Gradle, provided developers with sophisticated features, including code completion, debugging, and refactoring support. These tools, often tightly integrated with the Java compiler, significantly enhanced developer productivity and made Java an attractive choice for software development.

1. Challenges:

It's important to note that the challenges faced by the latest version of the Java compiler may vary over time, and new challenges may arise with the introduction of new language features or advancements in the Java platform.

As of data we found cutoff in September 2021, the latest version of the Java compiler is Java 16. While the Java compiler has undergone significant advancements over the years, it still faces some challenges. Here are a few common challenges that the latest version of the Java compiler might encounter:

=> Language Complexity:

Java is a mature and feature-rich language with a large set of language constructs, libraries, and APIs. Maintaining backward compatibility while introducing new language features can be a challenge for the Java compiler.

=> Performance Optimization:

The Java compiler strives to generate efficient bytecode that can be executed with high performance. However, optimizing the compilation process and generating highly optimized machine code can be challenging. The Java compiler needs to balance between compilation time and the quality of generated code to ensure that the compiled Java applications perform well.

=> Language Evolution:

As the Java language evolves, new language features and syntax enhancements are introduced. Incorporating these new features into the Java compiler can be a challenge, especially if they require changes to the existing compiler infrastructure.

=> Compatibility and Interoperability:

The Java platform aims to provide compatibility across different versions and implementations. The Java compiler faces the challenge of ensuring that the compiled bytecode is compatible with various Java Virtual Machine (JVM) implementations. It must adhere to the Java Language Specification (JLS) and correctly handle compatibility issues that may arise due to changes in the language or runtime environment.

=> Performance Profiling and Debugging:

As Java applications become larger and more complex, efficient profiling and debugging capabilities are crucial.

=> Tooling Support:

The Java compiler is closely integrated with various development tools, such as IDEs, build systems, and code analysis tools. Ensuring smooth integration and compatibility with these tools can be challenging, especially when new language features or compiler optimizations are introduced.

=> Performance of Incremental Compilation:

Incremental compilation, where only modified source files are recompiled, is important for efficient development workflows. Ensuring that the Java compiler fcan quickly and accurately determine the necessary recompilation dependencies and perform incremental compilation with minimal overhead is an ongoing challenge.

It's important to note that the challenges faced by the latest version of the Java compiler may vary over time, and new challenges may arise with the introduction of new language features or advancements in the Java platform.

**TABLE 2 [6]**

**Versions and Release Date of Java.**

|  |  |  |
| --- | --- | --- |
| **Java SE Version** | **Version Number** | **Release Date** |
| JDK (Oak) | 1.0 | January 1996 |
| JDK | 1.1 | February 1997 |
| J2SE  (Playground) | 1.2 | December 1998 |
| J2SE  (Kestrel) | 1.3 | May 2000 |
| J2SE  (Merlin) | 1.4 | February 2002 |
| J2SE  (Tiger) | 1.5 | September 2004 |
| Java SE 6  (Mustang) | 1.6 | December 2006 |
| Java SE 7  (Dolphin) | 1.7 | July  2011 |
| Java SE 8 | 1.8 | March  2014 |
| Java SE 9 | 9 | September, 21st 2017 |
| Java SE 10 | 10 | March, 20th 2018 |
| Java SE 11 | 11 | September, 25th 2018 |
| Java SE 12 | 12 | March, 19th 2019 |
| Java SE 13 | 13 | September, 17th 2019 |
| Java SE 14 | 14 | March, 17th 2020 |
| Java SE 15 | 15 | September, 15th 2020 |
| Java SE 16 | 16 | March, 16th 2021 |
| Java SE 17 | 17 | September, 14th 2021 |
| Java SE 18 | 18 | March, 22nd 2022 |
| Java SE 19 | 19 | September, 20th 2022 |
| Java SE 20 | 20 | March, 21st 2023 |

1. **FUTURE ADVANCEMENT IN JAVA COMPILER CONSTRUCTION**
2. *Introduction*

=> Briefly introduce the purpose of the paper and the context of the laboratory session.

=> Highlight the importance of teaching students about the technology involved with the Java programming language.

1. *Background*

=> Provide an overview of the Advanced Programming course and its objectives.

=> Explain the significance of understanding JVM architecture and how it relates to Java's platform independence.

1. *VisualJVM:* An educational software tool

=> Describe VisualJVM and its features.

=> Explain how VisualJVM provides a graphical front-end to the Java virtual machine.

=> Highlight the benefits of using VisualJVM as an educational tool.

1. *Laboratory Session Design*

=> Detail the structure and goals of the laboratory session.

=> Explain the specific activities and exercises that were conducted using VisualJVM.

=> Discuss the rationale behind using VisualJVM to teach JVM architecture and Java's platform independence.

1. *Student Reaction and Feedback*

=> Present the positive reactions and feedback received from the students.

=> Include quotes or anecdotes from students regarding their experience with VisualJVM.

=> Discuss the implications of the positive student reaction for future use of VisualJVM in different contexts.

1. *Conclusion*

=> Summarize the main points discussed in the paper.

=> Emphasize the value of using VisualJVM to enhance student learning in the context of the Advanced Programming course.

=> Discuss potential future directions for the use of VisualJVM or similar tools in programming education.

The JastAdd Extensible Java Compiler is a robust Java compiler that offers the flexibility to extend its functionality for various purposes, such as developing static analysis tools for Java and introducing new language constructs to Java. It follows a modular approach, starting with a Java 1.4 compiler and expanding it into a Java 5 compiler. One of its key features is the ability to handle complex extensions that impact name and type analysis effectively.

The compiler employs JastAdd, a declarative language similar to Java, as its implementation language. This choice enables a concise and clear description of the compiler architecture and facilitates easy extension development. Several example applications have been built using JastAdd extensions, including a Jimple code generator as an alternative backend, support for AspectJ constructs, and the implementation of a pluggable type system for non-null checking and inference.

In terms of performance, the extensible compiler demonstrates remarkable quality, speed, and size when compared to other frameworks for extensible Java compilers. It also outperforms traditional non-extensible Java compilers in terms of quality and size, while being only three times slower than javac, a widely used Java compiler.

The paper introduces the Extended Static Checker for Java (ESC/Java), which is an experimental compile-time program checker aimed at detecting common programming errors. ESC/Java uses verification-condition generation and automatic theorem-proving techniques to analyze annotated software.

The checker provides programmers with a simple annotation language that allows them to express design decisions formally. By examining the annotated software, ESC/Java identifies inconsistencies between the recorded design decisions in the annotations and the actual code. Additionally, it warns of potential runtime errors in the code.

The paper also mentions that the checker has been applied to tens of thousands of lines of Java programs, suggesting that it has been extensively tested and evaluated.

JADE (Java Agent Development Framework) is a software environment designed to create agent systems for managing networked information resources while adhering to the FIPA specifications for interoperable multi-agent systems. It offers a middleware that facilitates the development and execution of agent-based applications capable of seamlessly operating and interconnecting in both wired and wireless environments. JADE also supports the development of multi-agent systems through its pre-defined, programmable, and extensible agent model, along with a range of management and testing tools. Currently, JADE is one of the most widely used and promising frameworks for agent development. It boasts a large user community with over two thousand active members, has been employed in real-world systems across various application sectors, and its future development is overseen by a governing board comprising significant industrial companies.

The paper discusses SCOOP, a minimal extension to the sequential object-oriented programming model that enables concurrency. SCOOP introduces the "separate" keyword, which eliminates the need for explicit thread declarations, synchronized blocks, and explicit waits. It also ensures the absence of data races and atomicity violations through a set of compiler rules.

The paper's first contribution is a design pattern for SCOOP that allows the concepts of SCOOP to be applied to different object-oriented programming languages. This design pattern facilitates the transfer of SCOOP's ideas to other languages.

The second contribution of the paper is an implementation of the SCOOP design pattern for Java, demonstrating the generality of the SCOOP model. The authors also describe the tools that support the SCOOP design pattern and provide a concrete example of its usage in Java

The Java language is a versatile and widely used programming language that can be used for both front-end and back-end development. While Node.js is specifically designed for server-side JavaScript execution, Java provides a robust environment for building web applications using different technology stacks.

In the early days of web development, Java was primarily used on the back-end, while technologies like HTML, CSS, and JavaScript were used for front-end development. Java offered a multi-threaded environment and a wide range of libraries and frameworks for server-side development, making it a popular choice for building enterprise-level web applications.

Java's strengths lie in its scalability, performance, and extensive ecosystem. It supports multi-threading, allowing concurrent execution of tasks, which is beneficial for handling heavy workloads and improving response times. Java also has a vast array of libraries and frameworks, such as Spring, Hibernate, and Apache Struts, which provide powerful tools for building robust and scalable web applications.

When it comes to porting Java outside of the traditional server-side environment, there have been various projects and frameworks. For example, Apache Tomcat is a popular web server and servlet container that allows Java-based web applications to be deployed and run. Additionally, projects like GraalVM and Quarkus have emerged, enabling Java applications to be compiled into native executables, making them more efficient and portable

1. **ACKNOWLEDGMENT**

I would like to take this opportunity to thank everyone who helped me with this research. Throughout the research, I am grateful for their aspiring guidance, invaluable constructive criticism, and friendly advice. I am grateful to them for sharing their honest and illuminating perspectives on a variety of research-related issues

REFERENCES

[1] Oracle Website (Beginning of Java Compiler)

[2] Oracle Website (Design goals of Java)

[3] <https://en.wikipedia.org/wiki/Java_(programming_language)#:~:text=Java%20was%20originally%20developed%20by,by%20Sun%20under%20proprietary%20licenses>.

[4] Arthur van Hoff.

[5] <https://www.geeksforgeeks.org/phases-of-a-compiler/>

[6] <https://www.codejava.net/java-se/java-se-versions-history#:~:text=This%20article%20gives%20you%20an,Java%20version%20on%20your%20computer>).

[7] <https://ieeexplore.ieee.org/abstract/document/4448424>

[8] <https://dl.acm.org/doi/abs/10.1145/1297027.1297029>

[9] <https://dl.acm.org/doi/abs/10.1145/512529.512558>

[10] <https://link.springer.com/chapter/10.1007/0-387-26350-0_5>

[11] <https://ebooks.iospress.nl/doi/10.3233/978-1-60750-065-0-7>

[12] <https://ieeexplore.ieee.org/abstract/document/9424850>

[13] Comprehensive list of references including the Java Language Specification documents, technical articles, and related literature on the evolution of the Java front-end specification.