JOHANNES KEPLER
UNIVERSITY LINZ

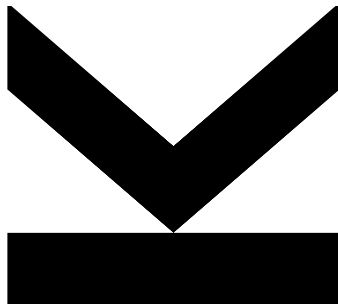
Author
Felix Schenk, BSc

Submission
Institute for System Software

Thesis Supervisor
a.Univ.-Prof. Dipl.-Ing. Dr. Herbert Prähofer

July 2024

Development of a Java Debugger Framework Based on the Espresso VM and Its Compilation to JavaScript



Master's Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstr. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Acknowledgments

First and foremost, I would like to thank Prof. Herbert Prähofer for his helpful guidance and encouragement throughout the supervision of my thesis. Additionally, I would like to thank Simon Grünbacher for the countless discussions and creative input. In general, I consider myself very fortunate to have worked with the JavaWiz development team, who provided invaluable feedback and support throughout the project.

Furthermore, I would like to express my gratitude to the team of Oracle Labs, especially Raphael Mosaner for coordinating communication and providing prompt technical advice. I also want to thank Patrick Ziegler, Aleksandar Prokopec, Gilles Duboscq and Allan Gregersen for their assistance with technical questions related to Espresso and WebImage.

Last but not least, I am immensely thankful to my mother, Iris Schenk, and my step-father, Markus Forster, for their relentless support throughout my studies. Your encouragement has been pivotal in my journey. Thank you.

This work has been funded by Oracle Labs within the research project *Java VM Compiler Performance*, subproject *GraalVM: JavaWiz Visualization Tooling*.

Abstract

JavaWiz is a visual debugger for Java designed to aid programming novices and programming teachers by providing various visualizations. These visualizations show intermediate program states. Currently, JavaWiz is distributed as a *Visual Studio Code* plugin, requiring users to download the appropriate software and setting up Java.

The goal of this thesis is to make JavaWiz accessible via a website. Thus, with this thesis, a new version of JavaWiz has been implemented which allows running JavaWiz in a web browser. First, a new backend of JavaWiz has been developed based on the *Espresso VM* by Oracle and the *Truffle Debug API*. Then, this new backend has been compiled to JavaScript using *WebImage*.

The thesis presents the implementation of the JavaWiz backend based on Espresso, the necessary modifications for compiling it with WebImage, an approach for handling multiple files with the web version, an approach for loading JavaWiz with predefined examples and a comparison of the performance of the different versions of JavaWiz.

Zusammenfassung

JavaWiz ist ein visueller Debugger für Java, der Programmieranfängern und -lehrenden durch verschiedene Visualisierungen helfen soll. Diese Visualisierungen zeigen Zwischenzustände des Programms. Derzeit wird JavaWiz als Plugin für *Visual Studio Code* bereitgestellt, weswegen Nutzende die entsprechende Software herunterladen und Java einrichten müssen.

Das Ziel dieser Arbeit ist es, JavaWiz über eine Website zugänglich zu machen. Daher wurde in dieser Arbeit eine neue Version von JavaWiz implementiert, die es ermöglicht, JavaWiz in einem Webbrowser auszuführen. Zunächst wurde ein neues Backend von JavaWiz implementiert, das auf der *Espresso VM* von Oracle und der *Truffle Debug API* basiert. Dann wurde dieses neue Backend mithilfe von *WebImage* zu JavaScript kompiliert.

In dieser Arbeit werden die Implementierung des JavaWiz-Backends auf der Grundlage von Espresso, die für die Kompilierung mit WebImage erforderlichen Anpassungen, ein Ansatz für die Handhabung mehrerer Dateien in der Web-Version, ein Ansatz zum Laden von JavaWiz mit vordefinierten Beispielen und ein Laufzeitvergleich der verschiedenen Versionen von JavaWiz beschrieben.

Table of Contents

1 Introduction **1**
1.1 Goals of this Work . . . 2
1.2 Structure . . . 3

2 Background **5**
2.1 JavaWiz . . . 5
2.2 GraalVM . . . 8
2.2.1 Compiler . . . 9
2.2.2 Truffle . . . 9
2.2.3 WebImage . . . 11

3 Related Work **13**
3.1 Java Visualizer . . . 13
3.2 Java Program Flow Visualizer . . . 14

4 Approach **16**
4.1 Espresso as a VM . . . 17
4.2 Backend in JavaScript . . . 18
4.3 Handling Multiple Files . . . 18

5 Espresso Backend **21**
5.1 Launching and Executing . . . 21
5.2 Debugging and Stepping . . . 22
5.3 Building Trace States . . . 26
5.3.1 Extracting the Call Stack and the Heap . . . 28
5.3.2 Extracting Loaded Classes and Statics . . . 35
5.3.3 Standard I/O Stream Handling . . . 36
5.3.4 Extracting Input Buffer Information . . . 37

6 JavaScript Backend **40**
6.1 Architectural Changes . . . 40
6.1.1 Sending Requests through a Shared Array Buffer . . . 43
6.1.2 Debugging Espresso after its Compilation with WebImage . . . 44
6.2 Substituting File I/O . . . 46
6.3 Supporting Reflection . . . 48

7 Handling Multiple Files **50**
7.1 Extending the Frontend 51
7.1.1 Storing Multiple Files in the Frontend 51
7.1.2 Reacting to File Changes 52
7.2 Adapting the Backend 53

8 Loading Predefined Examples **55**
8.1 Loading a Predefined Configuration 55
8.2 Example Website 58
8.2.1 Greatest Common Divisor 60
8.2.2 Read Integer 60
8.2.3 Matrix Multiplication 61
8.2.4 Integer List 62

9 Evaluation **64**
9.1 Hello World 65
9.2 Quicksort 65
9.3 List of Persons 66
9.4 Minesweeper 66
9.5 Object Array 68

10 Limitations and Outlook **69**
10.1 General Limitations 69
10.2 Future Improvements 70

11 Conclusion **71**

References **72**

Appendix **79**

Chapter 1

Introduction

This thesis builds on and extends JavaWiz [1], which is developed at the *Institute for System Software (SSW)* at the Johannes Kepler University, Linz. At its core, JavaWiz is a visual debugger and is currently distributed as a *Visual Studio Code (VS Code)* extension. It features various visualizations, including a desk test [1] depicting changes of variables over time, a graph of the call stack and the heap [1], a flowchart [2], special renderings for data structures, like arrays, linked lists [3] and trees [3] and an input buffer visualization. In cooperation with *Oracle Labs*, JavaWiz is continuously extended and improved.

The frontend of JavaWiz is written in *TypeScript* [4] with the framework *Vue.js* [5]. For the dynamic and interactive visualizations, *D3.js* [6] is used, enabling data-driven graphics and animations between different program states. JavaWiz' backend is written in *Kotlin* [7], which is compiled to Java bytecode. The frontend and backend adhere to a client-server architecture and communicate over a *WebSocket* [8] interface [1].

In addition to the VS Code extension, we want to make JavaWiz available to users via a website. This would reduce the complexity of setting it up and JavaWiz would be easily accessible. Because of the current architecture, the server would have to run the code provided by users. Consequently, the server hosting JavaWiz would be required to have extensive resources, and the system would have to be secure enough in order to execute untrusted code by users.

One way to overcome these issues is to run and debug the Java code provided by users on the

client side. This can be accomplished by utilizing the *GraalVM* [9] by Oracle, which makes it possible to run Java code in a JavaScript runtime. This way, the Java code is executed by the client's browser's engine, reducing workload and security concerns when hosting JavaWiz.

GraalVM is a high-performance polyglot *Virtual Machine (VM)* written in Java [10]. It features *Truffle* which seamlessly composes different language implementations in a single multi-language runtime [11]. Truffle provides a framework for implementing language interpreters defined as self-optimizing *abstract syntax tree (AST)* interpreters [12]. There are Truffle implementations for JavaScript, Python, Ruby, R, LLVM languages and many more [10, 12, 13]. Additionally, Truffle is capable of handling Java bytecode with *Espresso*, a full *Java Virtual Machine (JVM)*, which allows Java bytecode to be executed within another JVM in a meta-circular fashion [13].

Additionally, GraalVM features the creation of native images, *Ahead-Of-Time (AOT)* compiled native executables [10, 13]. Together with *Substrate VM* [13], an embeddable, precompiled VM, Java applications can be compiled with *Native Image* to mitigate startup time and enhance performance in general. On top of that, GraalVM offers *WebImage*, an AOT compiler cross-compiling Java bytecode to JavaScript utilizing various optimizations and control flow reconstruction [14].

This was only a brief overview of the GraalVM. A more detailed explanation, including descriptions of the just mentioned features and aspects, will follow in Section 2.2.

1.1 Goals of this Work

Combining both JavaWiz and GraalVM allows executing code provided by the user on the client side. Using Espresso and WebImage from GraalVM, the goals of this thesis are:

1. Instead of utilizing a separate JVM process, JavaWiz should employ Espresso to directly run and debug the provided code within the backend.

2. With the help of WebImage, the backend with Espresso as a VM should be compiled to JavaScript and distributed alongside the frontend.

3. The web version of JavaWiz should be extended to support multiple source files and text files.

As a result, the dependency on a system-side JVM is removed and Java code is executed on the client side within a JavaScript runtime environment. Figure 1 provides a general overview of what we want to accomplish. The upper part outlines the architecture of the first goal. The user enters Java source code in the frontend depicted on the left. Upon clicking the start button, the source code is sent to the backend via a WebSocket connection. With the current version of JavaWiz, the backend would start a VM in a separate process and communicate with it through the *Java Debug Interface (JDI)* [1]. By employing Espresso, as illustrated in the figure, the source code can be executed and debugged directly in the backend. During debugging, intermediate states are sent to the frontend to be visualized. These states form a trace in the frontend which the user can navigate. Following the user demanding a new step, the frontend sends a request to the backend. Consequently, the user code is executed there until the requested step has concluded. Then, all collected trace states are sent to the frontend.

The architecture of the second goal is shown in the lower part of the figure. Using WebImage, the JavaWiz backend with Espresso is compiled to JavaScript. Moreover, the communication between frontend and backend has to be adapted to meet the requirements of the JavaScript environment. With this transformation, both the backend and the frontend can be executed directly in the user's browser's JavaScript engine.

In Chapter 4 we will show a more detailed view of these architectures.

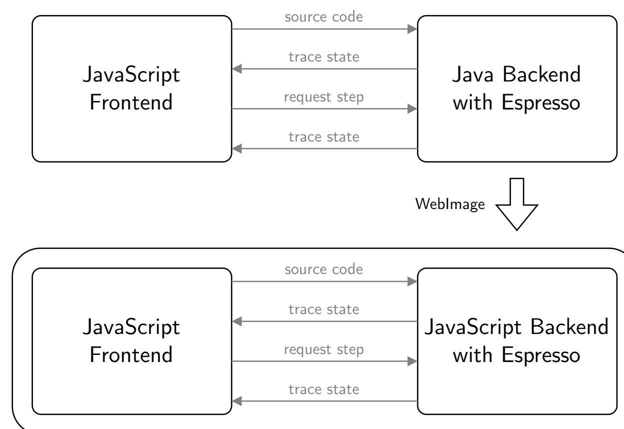


Figure 1: Overview of the Architecture of this Work.

1.2 Structure

This thesis is structured as follows:

- Chapter 2 further elaborates on JavaWiz and the GraalVM.
- Similar tools to JavaWiz are discussed in Chapter 3.
- The approaches to reach the goals of this thesis are covered in Chapter 4.
- Chapter 5 deals with employing Espresso in the JavaWiz backend.
- Compiling the backend to JavaScript using WebImage and associated modifications and challenges are discussed in Chapter 6.
- Chapter 7 covers extending the frontend and adapting the backend to support handling multiple files in the web version of JavaWiz.
- The system is demonstrated by presenting several examples in Chapter 8.
- Chapter 9 compares the different versions of the backend in terms of start-up time and runtime.
- Limitations and future developments are given in Chapter 10.
- The thesis presents a summary and a conclusion of the work in Chapter 11.

Chapter 2

Background

This chapter provides an overview of the systems and technologies this thesis is built on. First, the existing system JavaWiz is described. Following, we introduce GraalVM, the platform enabling this thesis.

2.1 JavaWiz

JavaWiz is a tool developed at the Institute for System Software that targets novice programmers. It functions akin to a debugger and additionally provides visualizations for the execution of Java programs. Originally initiated with the master's thesis of Katrin Kern [1], JavaWiz has undergone enhancements by various visualizations introduced by subsequent student projects and theses:

- Andreas Schlömicher implemented a flowchart component [2].
- Melissa Sen developed a visualization of the input stream and a sequence diagram component [15].
- Simon Grünbacher contributed to the development of the VS Code extension and various other improvements.
- Additionally, I created visualizations specifically tailored for linked lists [3], binary trees [3] and arrays.

As mentioned earlier, there exists a VS Code extension for JavaWiz. When installed, users can run Java applications with JavaWiz. Figure 2 presents a screenshot of the extension. The sections are described in the following:

① The built-in editor of VS Code is utilized to display and edit the source code. The line which will be executed next is highlighted in blue.

② Upon launching, the JavaWiz pane appears on the right-hand side of the editor. It includes a toolbar and space for visualizations.

③ JavaWiz offers multiple stepping controls allowing users to restart the program, step back one step, step over, step into and step out a method and run to a line in the code. The status indicates whether JavaWiz is replaying previous states or whether the debugger is providing new program states.

④ Users can show different visualizations with the icons in the toolbar. The screenshot displays the stack and heap visualization. Other options include a flowchart component, the desk test, an input stream visualization, an array component, a linked list visualization and a binary tree component.

⑤ Below both the editor and the JavaWiz panel, an input/output terminal is available. It serves to display the program's output and to enter user input.

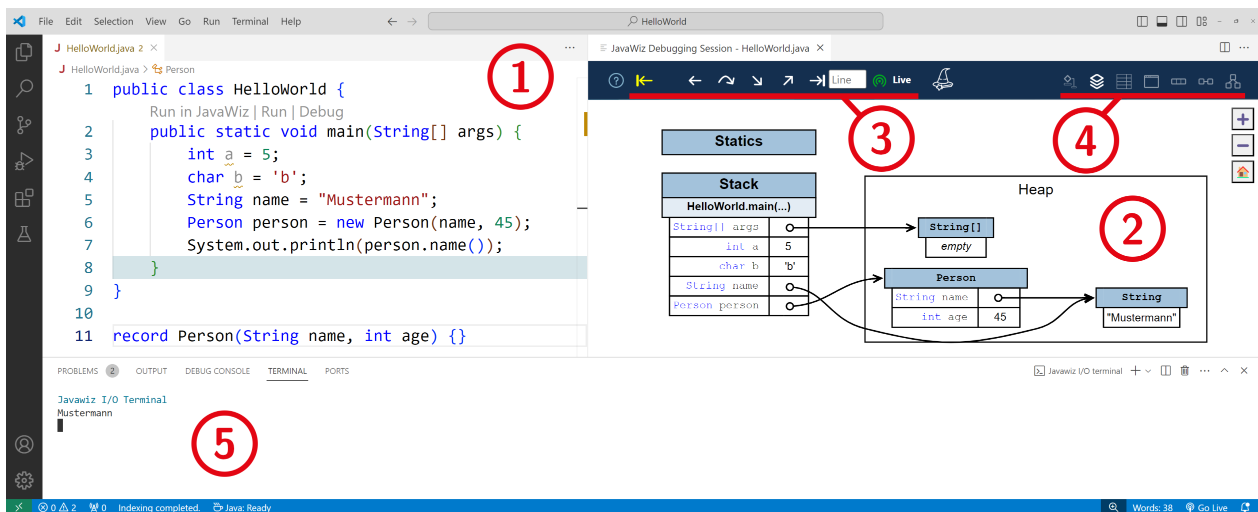


Figure 2: The JavaWiz Visual Studio Code Extension.

There exists a web version of JavaWiz in addition to the VS Code extension. It is accessible through a browser and incorporates a component with a text editor in place of the editor of VS Code. With this version, the backend is executed in a separate process. The main goal of this thesis is to provide a web-only version of JavaWiz. Thus, we will use the web version

for this thesis.

Why is the backend run in a separate process? To clarify that, let us look at JavaWiz' architecture shown in Figure 3. JavaWiz has two main components: the frontend and the backend. The former contains the code editor, a console and the visualizations (cf. Figure 2). The backend is responsible for parsing, modifying, compiling and executing the Java program. Frontend and backend communicate through a *WebSocket* interface in the following way:

1. When the user presses the compile and run button in the frontend (1), the source code is sent to the backend.

2. First, the backend parses and modifies the source code (2). As a result of modifications, some statements are instrumented for providing specific trace information, cf. Section 5.3.1. For instrumentation, the AST generated by the JavaParser [16] is used.

3. Next, the backend compiles (3) both the unmodified source code and the modified source code. Compiling the unmodified source code is done to send the user feedback about compile errors.

4. Then, a thread with a JVM (4) is started for debugging the class files of the modified source code (5). To extract state information, the JVM and the backend communicate through the JDI. Additionally, the backend dictates through the JDI when the VM should continue execution and in turn determines whether to send the collected program states to the frontend.

5. The VM (4) is suspended before the first statement in the main function. Its current state along with an AST generated by the parser (2) are sent to the frontend.

6. If visible, the frontend (1) visualizes the AST in the flowchart component and the first state of the program in the other visualizations.

7. When the user initiates a step request, it is forwarded to the backend.

8. The backend continues execution (4) until the request is fulfilled. The collected intermediate states are sent to the frontend (1).

9. Upon receiving the response of the backend, the frontend (1) processes the states. The states are stored and form a trace, which the user can navigate through. The latest

state received by the debugger is visualized.

10. Steps 7 to 9 may repeat until the end of the program is reached. Subsequently, the user is still able to browse through the history of states, but there is no communication between the frontend and backend anymore.

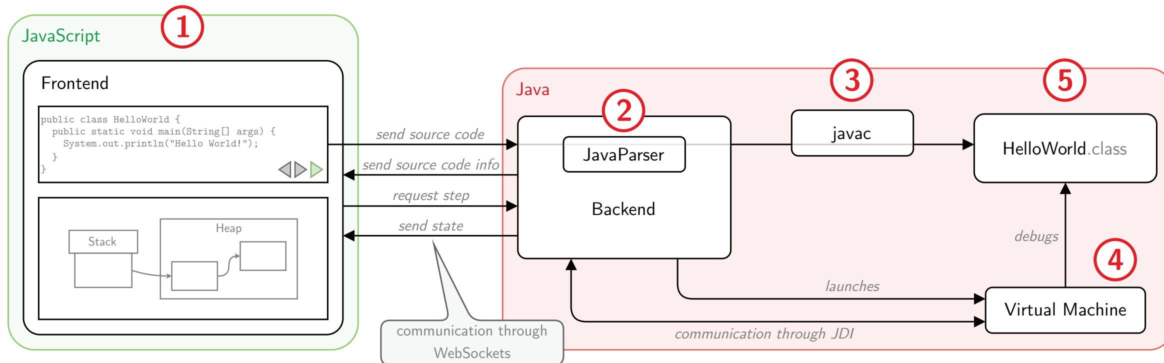


Figure 3: The Architecture of JavaWiz.

Chapter 4 will cover how this architecture is modified to achieve the goals defined in Section 1.1.

2.2 GraalVM

After outlining JavaWiz, we want to take a closer look at GraalVM, which has been briefly introduced in Chapter 1. GraalVM consists of various projects [9, 13] but altogether it is a high-performance *Java Development Kit (JDK)* distribution. It can be used to compile Java applications to native executables and additionally offers a polyglot VM. In the following sections, we want to give an overview of some projects within GraalVM:

- In Section 2.2.1 we want to briefly take a look at the compiler project. It performs aggressive optimizations on Java bytecode and compiles it to machine code [17].

- *Truffle* is a framework for the execution and interoperability of various programming languages in a single VM [11]. It will be the topic of Section 2.2.2.

- Cross-compiling Java to JavaScript can be performed by employing *WebImage* [14]. This will be outlined in Section 2.2.3.

For an in-depth introduction, see the papers cited in the following or the GraalVM website [18].

2.2.1 Compiler

The Graal compiler aggressively performs optimizations including but not limited to inlining, constant folding and escape analysis [10]. These optimizations are performed on the Graal *Intermediate Representation (IR)*, which is structured as a directed graph in static single assignment form, in which both the data flow and the control flow are represented [17].

Figure 4 shows a Graal IR of an if statement as an example (cf. [17]). The value for the variable `result` depends on which branch of the statement is executed. Since the graph adheres to static single assignment form, a `Phi` node is employed. With it, the value for the variable after the if statement can be determined.

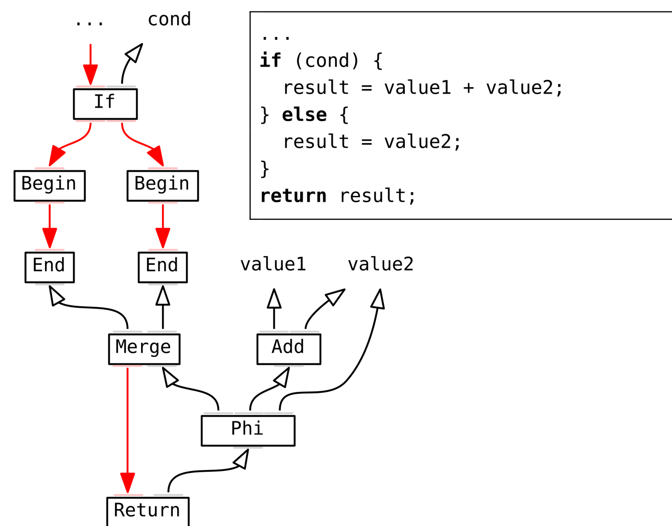


Figure 4: Example Graph of the Graal Intermediate Representation (taken from [17]).

When using Graal as JIT compiler, program execution works as the following (cf. Figure 5): The interpreter interprets bytecode and generates profiling information [19]. This information is used by the *Just-In-Time (JIT)* compiler to create machine code and deoptimization information in the course of compiling bytecode. Graal applies speculative optimizations. When an assumption is not met, the execution is transferred from optimized code back to the interpreter by using deoptimization.

2.2.2 Truffle

Truffle is a framework for implementing self-optimizing AST interpreters where a node can rewrite itself [20]. The Graal compiler compiles the interpreter together with the executed program and generates efficient machine code by applying *partial evaluation*. Let us take a look at the system structure for executing a guest language application in Figure 6 (cf. [20]).

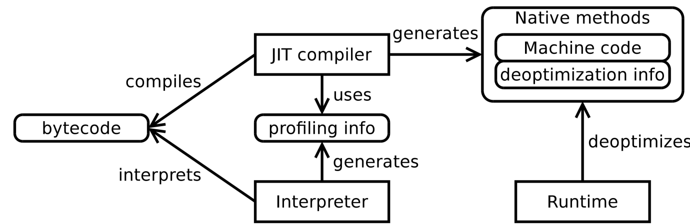


Figure 5: Graal Compiler System Structure (taken from [19]).

At the top, the guest language application is executed. It is written in the guest language by the application developer. Below it, the layer for the language interpreter is run. It is written by the language developer in a managed host language, in Truffle's case Java. All of this is dependent on the host services, also written in Java, which are processed by the OS.

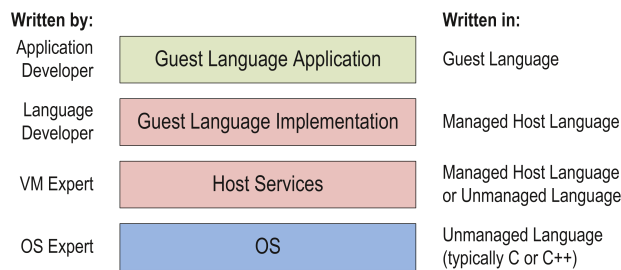


Figure 6: System Structure of Running a Guest Language Implementation (taken from [20]).

What makes for a self-optimizing AST interpreter? We want to explain this using Figure 7 (cf. [20]). It shows an AST consisting of uninitialized nodes. They are rewritten to generic nodes or to ones with concrete types by using profiling information. If executed repeatedly, the resulting AST is compiled using partial evaluation.

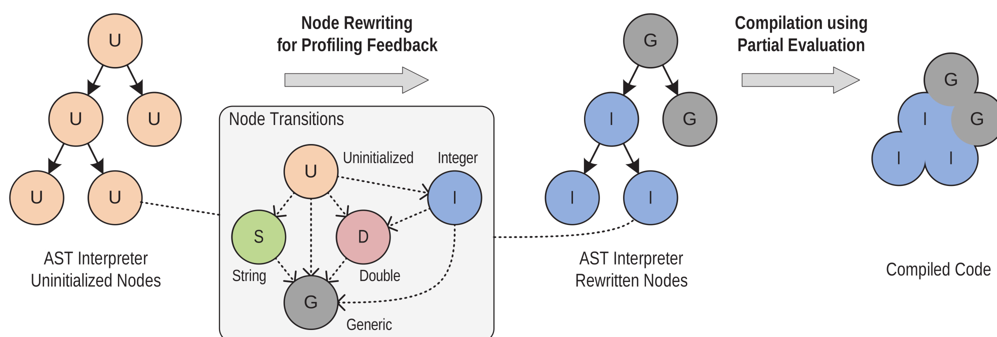


Figure 7: Example of Node Rewriting (taken from [20]).

If assumptions on type information turn out to be false during execution, deoptimization is triggered (cf. Figure 8) [20]. Subsequently, nodes are rewritten to match the updated type information. With these modifications, the AST may be compiled again.

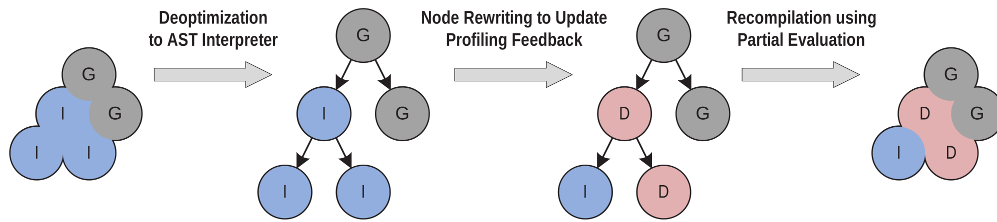


Figure 8: Example of Deoptimization After Node Rewriting (taken from [20]).

Various other optimizations are applied during the interpretation of such an AST, which mitigate the overhead of the dynamic interpretation [20]. Further, it is possible for Truffle languages to interoperate [11].

Espresso

This thesis builds on Espresso, which is a meta-circular JVM which uses the Truffle framework to interpret and execute Java bytecode [13]. It enables running code inside an isolated context and dynamically loading and running Java. In this thesis, Espresso is employed to debug Java code entered by a user of JavaWiz and will finally be run in a JavaScript runtime. Chapter 4 will explain this in detail.

To generate a trace state during execution, we need to access the internal state of Espresso. For this purpose, Truffle provides a Debug API. This API is language agnostic and provides debugging functionality for all Truffle languages. For example, it offers access to the stack and heap of the executed program. If language specific debugging information is required, the internal structure of a Truffle language has to be taken into account. Section 5.3 will cover how the state of Espresso is extracted.

2.2.3 WebImage

Cross-compiling Java to JavaScript can be achieved with WebImage [14]. WebImage builds upon Native Image and uses the Substrate VM to apply its static analysis technique to enable AOT compilation [21].

The system structure is depicted in Figure 9 (cf. [14]). Java bytecode is input to the system and statically analyzed to retrieve which classes and methods are reachable. This static analysis is required because compiling the entire Java code of an application including its libraries and a whole JDK would be infeasible. Subsequently, this information is fed to the Graal frontend which applies numerous optimizations and outputs a Graal IR. The Graal backend is replaced, since the IR should not be executed but rather converted to JavaScript

source code. For that, a structured control flow has to be reconstructed, since the Graal IR is unstructured. Next, the IR is used to generate JavaScript source code which can be run by a JavaScript VM.

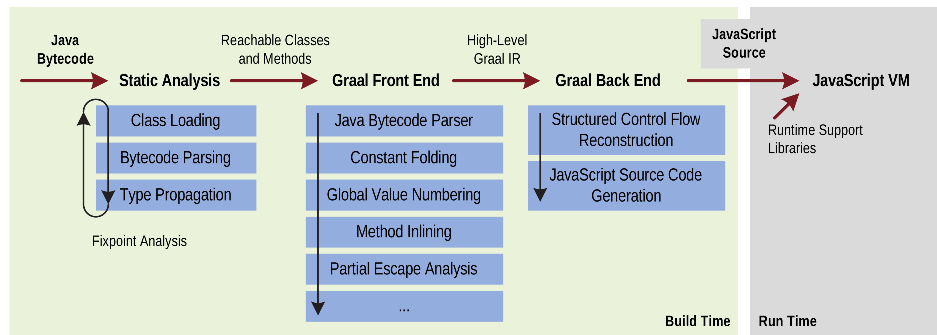


Figure 9: System Structure of WebImage (taken from [14].

Chapter 3

Related Work

For related work, we are looking for online Java debuggers which are capable of visualizing intermediate program states. Katrin Kern found several programs similar to the VS Code version of JavaWiz [1]. Here, we especially take a look at tools available via a website. In the following sections, we will introduce two related projects.

3.1 Java Visualizer

Java Visualizer [22] was developed by David Pritchard and Will Gwozdz. It reuses the frontend of the Online Python Tutor [23] by Philip Guo. The Java adaption was developed by the Centre for Education in Mathematics and Computing at the University of Waterloo [24]. Both the website of the center [24] and the website of the Online Python Tutor [23] host the Java Visualizer, while the former has more features.

A screenshot of the tool is shown in Figure 10. The left side contains the visualized code and user controls for stepping. The visualization of the call stack and the heap is displayed on the right. With the help of the graphical representation, it is easier for programming novices to understand references, arrays, call stack generation and more. The version on the website of the Centre for Education in Mathematics and Computing [22] supports console input and command line arguments. It is important to note, that console input has to be specified before running the program.

The tool has some limitations [26]:

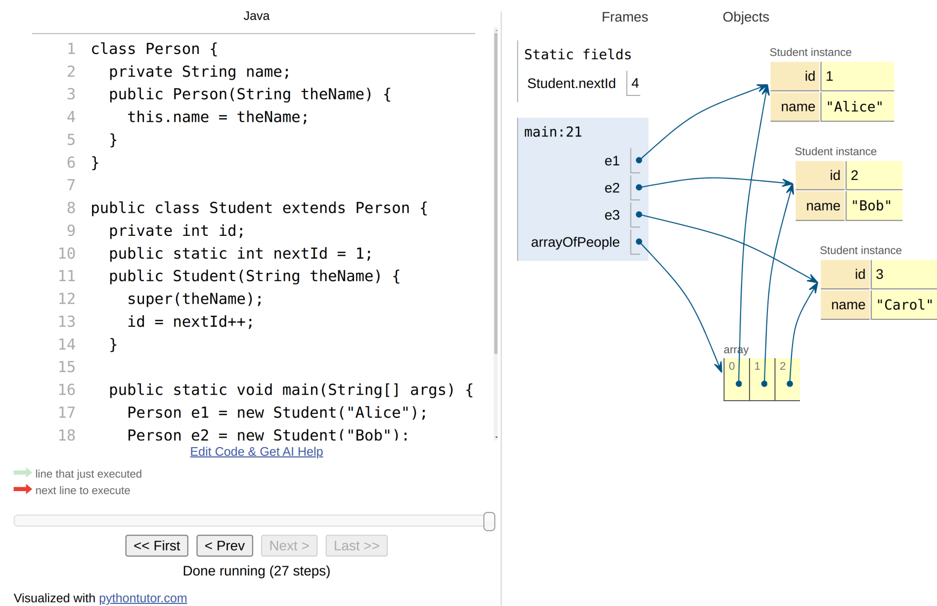


Figure 10: Screenshot of the *Java Visualizer* [25].

- Source code entered by users is sent to a server running the backend. It tries to execute the whole program and then sends data for visualizations back to the frontend. As mentioned in Section 1.1, one of the goals of this thesis is to execute the code on the client side. This leads to less load and improved security on the server hosting the tool. Additionally, JavaWiz allows running the program step-by-step, i.e., as far as the user requested. This means that new program states are generated in real time, and it also allows handling console input live.

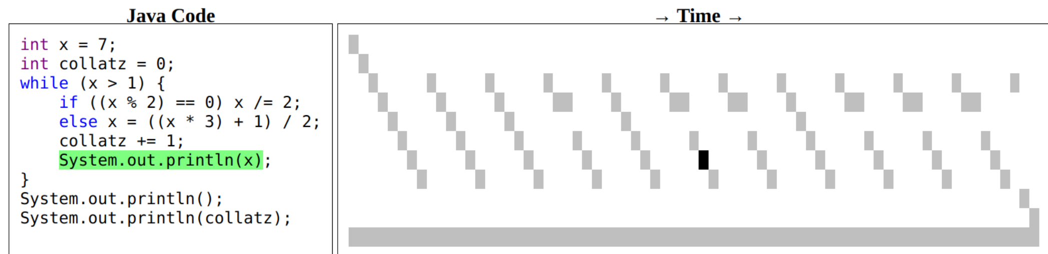
- Java Visualizer currently supports only Java 8, an outdated version of Java. JavaWiz currently supports Java 17, which is newer than Java 8. Since it is not the newest version, the JavaWiz team is trying to remove all dependencies constraining the Java version. Chapter 10 will give more insights on this.

- The online tool only offers executing a single source file, while the goal of this thesis is to support multiple source files as well as text input and output files.

3.2 Java Program Flow Visualizer

Another tool for visualizing Java code is the Java Program Flow Visualizer, written by Luther Tychonievich [27]. Similar to JavaWiz, the motivation to create it was to help students learning programming in a university course [28].

Figure 11 shows a screenshot of the Java Program Flow Visualizer. On the upper left side is the executed code. Next to it is a timeline showing when each line was executed. Below both is a visualization of the call stack and the program's output. Hovering over a point in time in the timeline shows the corresponding state of the stack and of the program output. The tool does only support a subset of Java [27], yet it runs fully on the client side. For that, it employs a Java interpreter written in JavaScript [28].



Mouse-over times to see how things change. The bottom row of the times is something like a slider, allowing you to more easily "run" the program.

Below is the memory and output as it looks after completing the highlighted part of the code.

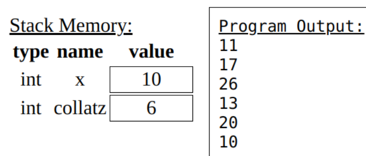


Figure 11: Screenshot of the *Java Program Flow Visualizer* [27].

Despite running Java code on the client side, the Java Program Flow Visualizer has some drawbacks:

- Since only a subset of Java is supported, the capabilities of the tool are limited. The system allows creating integers, doubles and booleans, yet there is no support for objects [27]. As a result, it particularly does not allow for exceptions or accessing the Java standard library. In contrast, objects and the Java standard library are supported by JavaWiz, as it utilizes a full JVM to run the code. Nevertheless, there are also some related limitations when using JavaWiz which we will discuss in Chapter 10.

- Because of the limited version of Java, the tool does not support multiple source files nor multiple methods [27]. Contrary, one goal of this thesis is to support multiple files.

- Reading console input or handling native calls in general is not supported by the Java Program Flow Visualizer [27]. In contrast, various native calls are supported by JavaWiz including reading from the standard input stream and file I/O, see Section 5.3.3 and Section 6.2.

Chapter 4

Approach

After introducing the used technologies and related work, this chapter proposes approaches on how to reach the goals set in Section 1.1. We want to briefly revisit them in the following:

- Removing the separate VM is the first step to fulfill this thesis' goals. Section 4.1 is about replacing the current VM with Espresso. As a result, Java programs are run and debugged directly within the backend.

- While this step removes complexity, the server would still have to provide enough resources to run all user's code, as it would have to run the backend. Section 4.2 covers how the backend is compiled to JavaScript with WebImage which enables it to be run in a browser. As a result, the server has to host the generated JavaScript file, but does not need to execute untrusted code anymore.

- The third objective, to extend JavaWiz to be capable of handling multiple files, will be covered in Section 4.3.

The three objectives extend the existing JavaWiz architecture, as introduced in Section 2.1. Recall that there are two parts, the frontend and the backend. The former is responsible for building a trace, visualizing states within it and handling user interaction. The backend is responsible for parsing, modifying, compiling and debugging code entered by users. The latter is done by a JVM running in a separate process.

4.1 Espresso as a VM

This section covers replacing the existing VM with Espresso. The proposed architecture is shown in Figure 12. Changes to JavaWiz' architecture in Figure 3 of Section 2.1 are emphasized with bold fonts and thicker lines. Employing Espresso leads to several changes:

1. The most significant change is the utilization of the Truffle Debug API to communicate with Espresso. It replaces the current dependence on the JDI. As covered in Section 2.2.2, the API exposes the current state of debugged Truffle executions. Thus, it provides access to the state of the program running in Espresso.

2. Launching a main function with Espresso differs from launching it with a VM in a separate process. However, this change has relatively little impact, compared to the other modifications. To achieve a similar architecture, the backend runs in multiple threads. One thread determines if Espresso should continue execution and communicates with the frontend. Another thread executes Espresso and extracts its state information.

3. The debugging process of Espresso vastly deviates from that of a system VM. Although Espresso is debugged using the Truffle Debug API, it does not block itself from resuming execution. It continues running after the current suspension has been handled. Consequently, the logic to block Espresso and await user input must be implemented manually.

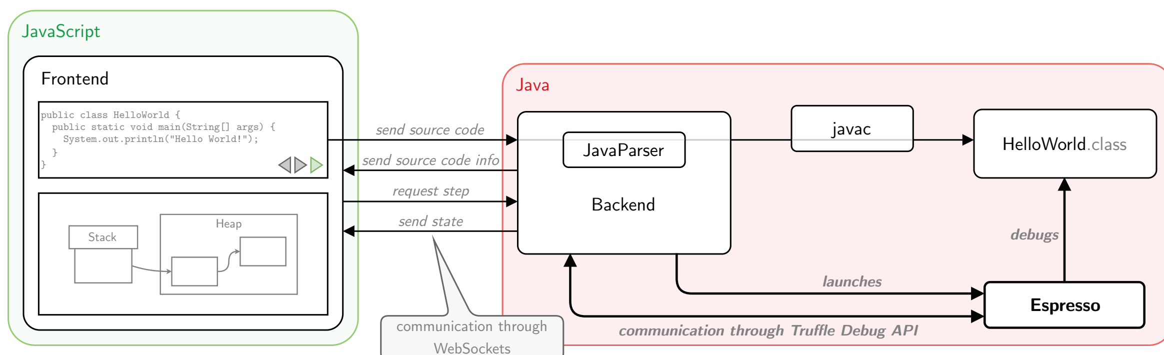


Figure 12: The Architecture of JavaWiz With Espresso as a VM.

Chapter 5 covers replacing the VM with Espresso in detail.

4.2 Backend in JavaScript

After enabling debugging of user code directly within the backend, the next step is to compile the backend to JavaScript with WebImage. Figure 13 shows the proposed architecture to accomplish this. The three components written in Java, the backend, the compiler and Espresso, are compiled to JavaScript using WebImage. Besides being executed by JavaScript, the fundamental architecture does not change.

This approach has the following challenges:

1. The entire backend is executed within a single thread. As a result, both the backend logic and Espresso have to run within the same thread. Consequently, this thread has to fulfill what was previously done with two threads. It has to determine when to transmit the collected trace states to the frontend, handle the communication with the frontend in general and run Espresso.

2. Furthermore, special handling is required for native calls. Reading the standard input stream and native file I/O is not supported by JavaScript.

3. Additionally, the communication between the frontend and backend has to be changed from WebSockets to the Web Worker interface [29] and shared memory [30].

4. The backend uses reflection for parsing and creating *JavaScript Object Notation (JSON)* objects. These are used for communicating with the frontend. Moreover, the JavaParser [16], used for parsing and modifying the user code, uses reflection. Since WebImage does not automatically detect all reflection cases, specific treatment is required [21]. In detail, a configuration file defining the usages of reflection has to be provided [9].

Details of compiling the backend to JavaScript with WebImage are presented in Chapter 6.

4.3 Handling Multiple Files

The current web version of JavaWiz only supports editing a single source file. This leads to the third goal of this thesis: JavaWiz should be able to handle multiple files, including Java source files as well as text input and output files.

Figure 14 illustrates how multiple files are handled. The frontend stores multiple files and

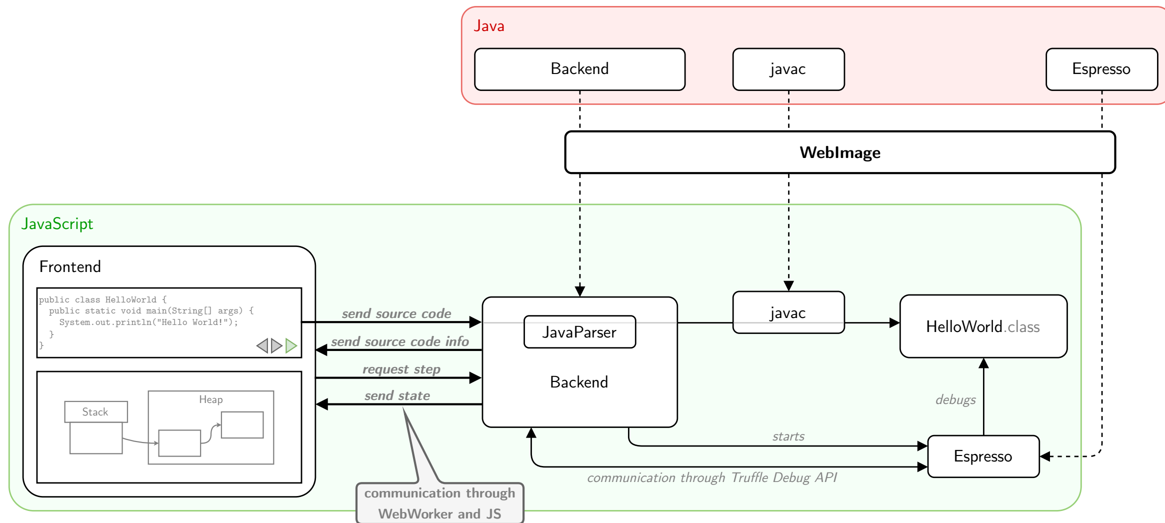


Figure 13: The Architecture of JavaWiz with Espresso as a VM and Compiling it With WebImage.

whenever the user requests execution, the files are sent to the backend. There they are split into source files and text files. Since JavaScript has no native access to the file system, WebImage provides an in-memory file system. Source files are stored in a source directory, while text files are placed inside the working directory of Espresso. This way, text files are accessible by the user's program.

Employing the in-memory file system simulates a conventional file system. As a result, JavaWiz works similarly to native code editors. Yet, there is one problem with this solution: The files accessible to Espresso are only stored in the backend. Thus, the user would not be able to observe file changes or creations. As a solution, the backend notifies the frontend about file modifications, allowing the current state of the files to be exposed to the user. It should look as if the backend has access to the files opened in the frontend, while these files are actually just mirrored.

The current backend is already able to handle multiple source files, as this has been a requirement for the VS Code version. Consequently, the backend does not need to be modified to support multiple source files. The main changes in the backend concern handling text files.

Chapter 7 covers how this approach was implemented.

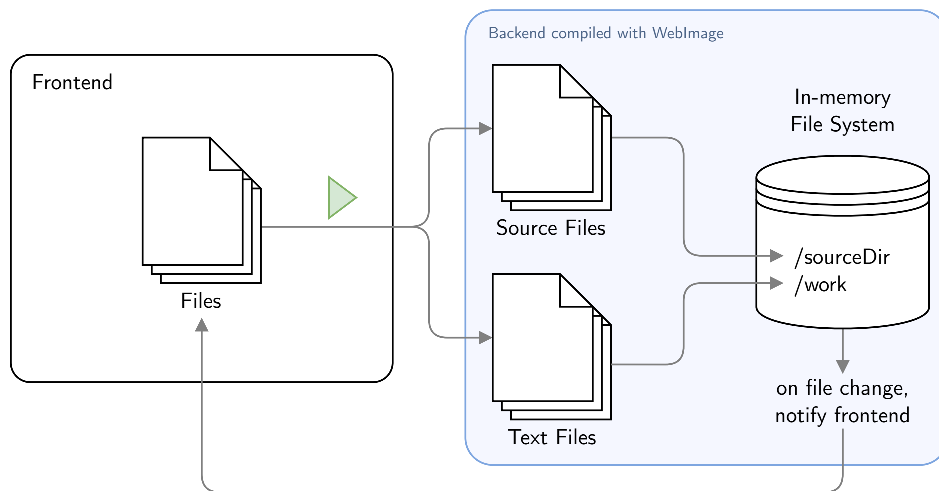


Figure 14: Approach for Handling Multiple Files.

Chapter 5

Espresso Backend

This chapter is about using Espresso as a VM in the backend. It builds upon the architecture presented in Section 4.1 and should function the same way as with a VM in a separate process. The following sections will explain how this is accomplished:

1. Launching Espresso and executing Java code is covered in Section 5.1.
2. Section 5.2 presents how Espresso is debugged and how blocking and stepping is implemented.
3. Building trace states is described in Section 5.3.

5.1 Launching and Executing

This section describes the process of launching Espresso and executing the main function. Here is an overview about the process:

1. Truffle languages are run within contexts [31]. This allows isolating guest languages, defining their permissions in terms of access to the host system and setting additional options. Hence, a context for Espresso is created first.
2. Then, Java is initialized.
3. The class containing the main function is loaded.

4. Finally, the main function is invoked.

Listing 1 shows a simplified version of the code accomplishing this. The actual code mainly differs in setting the options of the context builder. In lines 1 to 5 a context is constructed with a `Context.Builder`. When creating a context, various options and limitations may be set [32]. With the class path of Java, a language-specific option is set in line 2. Additionally, it is possible to set the standard output, error and input streams of the guest language, cf. lines 3 to 5. Following that, the context gets built in line 6.

With line 7 Java gets initialized within its context. The returned bindings represent a class loader. Subsequently, the class containing the main function is queried in line 8. The main function is invoked by providing its signature in class file notation, as depicted in lines 9 and 10. The passed arguments, in this case an empty string array, have to be internal Espresso objects. We will omit how these are created here.

```
1  val builder = Context.newBuilder()
2    .option("java.Classpath", cp)
3    .out(outputStream)
4    .err(errorStream)
5    .`in`(inputStream)
6  val context = builder.build()
7  val bindings = context.getBindings("java")
8  val mainClass = bindings.getMember(fullyQualifiedMainClassName)
9  val returnValue = mainClass
10   .invokeMember("main/([Ljava/lang/String;)V", emptyStringArray)
```

Listing 1: Launch Espresso and Execute Main Function (simplified).

5.2 Debugging and Stepping

So far, we know how to start Espresso and launch a main function. The next step is to debug the execution of Espresso.

To initiate debugging Espresso with the Truffle Debug API, a `DebuggerSession` has to be started. Since we want to capture every state of the program, the session has to be initiated before launching the main function. In Listing 1 this would be done right after line 7. Listing 2 shows how to start a debugger session. Line 1 finds a `Debugger` which is bound

to a context's engine. Subsequently, a debugger session is started in line 2. The parameter for the function `startSession` is a `SuspendedCallback`. This callback is invoked whenever Espresso is suspended. It receives a `SuspendedEvent`. The event serves two important tasks: (1) defining when the next suspension is supposed to happen and (2) extracting current state information. Both can be done within the callback in place of line 3. As a result, handling the suspended event enables introducing additional logic for debugging.

```
1 var session = Debugger.find(context.engine)
2   .startSession { suspendedEvent ->
3     // handle suspended event
4   }
```

Listing 2: Starting a Debugger Session.

After introducing how to start a session, we want to show how it impacts execution. Figure 15 illustrates the flow of a debugged Espresso execution. The gray region highlights the execution of the main program, while the red areas depict executions of suspended callbacks. The main program starts ▷ and eventually hits a suspension point ○. Subsequently, the callback is called. After that, the execution of the program continues. Then another suspension point ○ is encountered. Thus, the suspended callback is triggered once more. Note that the suspended callback is run within the same thread that runs Espresso.

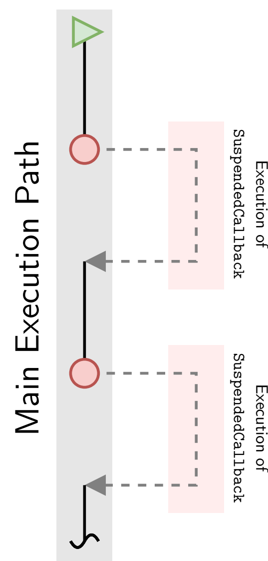


Figure 15: Flow of a Debugged Execution with Espresso

Now we have to address two missing points:

1. We have to determine, when to suspend Espresso. Otherwise, we would never trigger the suspended callback.

2. What to do in the callback. As mentioned earlier, this enables us to introduce additional logic in the debugging process.

There are three methods for suspending Espresso:

1. **Suspend via debugger session.** It is possible to suspend all threads or a single thread immediately, to suspend when a specific node is reached, or to suspend the next execution.

2. **Suspend via breakpoints.** Breakpoints can be set for a source file and optionally a line or a column in it. Additionally, one can define if the breakpoint suspends only the first time it is reached. Further, it is possible to specify how many times a breakpoint is ignored before causing a suspension and whether the suspension should happen before or after it. Exception breakpoints are also supported.

3. **Suspend via suspended event.** Within the suspension callback, decisions regarding the next suspension can be made. Options include stepping in, stepping over and stepping out of methods. Other choices are to continue execution until one of the other suspension methods halts it and to terminate the execution.

In our case, we want to suspend Espresso after each statement. The initial suspension should happen before executing the first statement in the main function. Thus, before invoking the main function, the subsequent execution is suspended via the debugger session.

The final point to address is the logic inside a suspended callback. This is the most important step, as it is the only way of enabling state extraction and incorporating custom logic into Espresso's debug execution. The necessary steps within the callback are:

1. **Extract state information.** Through the received suspended event, the callback gains access to the current program state. This includes the call stack, heap and other relevant information. This data needs to be combined to a trace state.

2. **Wait for a user request after concluding a step.** In Figure 15 we have seen that Espresso automatically continues execution after calling a suspended callback. Since we

want to halt Espresso after a step has been fulfilled, it must be blocked from continuing.

3. **Define next suspension.** Since we want to trace all steps, the next suspension point has to be set. Hence, a step-into is requested from each suspended event.

Figure 17 illustrates this process. For better understandability, we introduce an illustrative example in Figure 16. In the example a variable **x** is declared and initialized in (A). This variable is then incremented by one in (C) and multiplied by five in (E). Between these statements, we want to suspend Espresso. Thus, the relevant suspension positions are (B) and (D). Note that the execution will also be suspended before (A) and after (E).

```
int x = 5;  <-------  (A)
            --------  (B)
x = x + 1;  <-------  (C)
            --------  (D)
x = x * 5;  <-------  (E)
```

Figure 16: Example Code to Better Illustrate Figure 17.

Figure 17 shows a sequence diagram illustrating the execution flow of the example program. There are four object lifelines in the sequence diagram:

(I) The request handler is responsible for communicating with the frontend and deciding when to send the collected trace states.

(II) A semaphore is employed as a synchronization mechanism for stepping. It is used to block Espresso and resume it when requested by the user.

(III) A blocking queue is used to manage the synchronization of step results. Step results contain trace states along with information if the VM is still running, whether input is needed and of a possible Espresso exception. The queue is used to block the request handler while waiting for tracing information from Espresso.

(IV) The thread running Espresso is executing the user's code and additionally executes the suspended callbacks.

The sequence diagram starts with Espresso executing the first line of the example (A). Following the execution of this line, Espresso detects a suspension point (B). Hence, it triggers the execution of the suspended callback. Within this callback, the suspended event is handled

(1). As previously mentioned, the initial step within the callback involves extracting state information. This is achieved by building a trace state (2). Then, the trace state is wrapped in a step result and put into the step result queue (3).

It is important to note, that the request handler has been awaiting the state information of (A). Why? It permitted Espresso to continue execution in the suspended event before (A). Then, it requested a step result from the queue (4) and has since been waiting for it in a blocked state. Upon receiving the step result (5), the request handler is unblocked. Thus, it signals the completion of the step (6). Further, a decision is made whether Espresso has run far enough for the collected trace states to be sent to the frontend (not explicitly shown in the diagram).

Meanwhile, the Espresso thread requests the lock for executing the next step (7). Thus, further execution is blocked until another step is requested. A subsequent step is triggered either by user input or if the previous step request has not yet concluded. When another step is needed (8), the step lock is released by the request handler (9) and the Espresso runner is unblocked (10). The final step of the suspended callback is to define the next suspension point (11).

The next step's execution starts with the request handler requesting another step result from the queue (12). Then the process continues with steps (13) to (22) handling the next execution step in the same way as before.

5.3 Building Trace States

We now show how the extraction of state information works. The following data is extracted from the debugger and combined to a trace state:

- **Source file URI**. To enable the frontend to highlight the current line, it requires the information about which file is currently executed.
- **Line number**. In addition to the file URI, the frontend needs to know the line number where the debugger is currently suspended. This is used to highlight the line.
- **Stack frames**. These contain information about variables on the call stack. Their detailed extraction is discussed in Section 5.3.1.

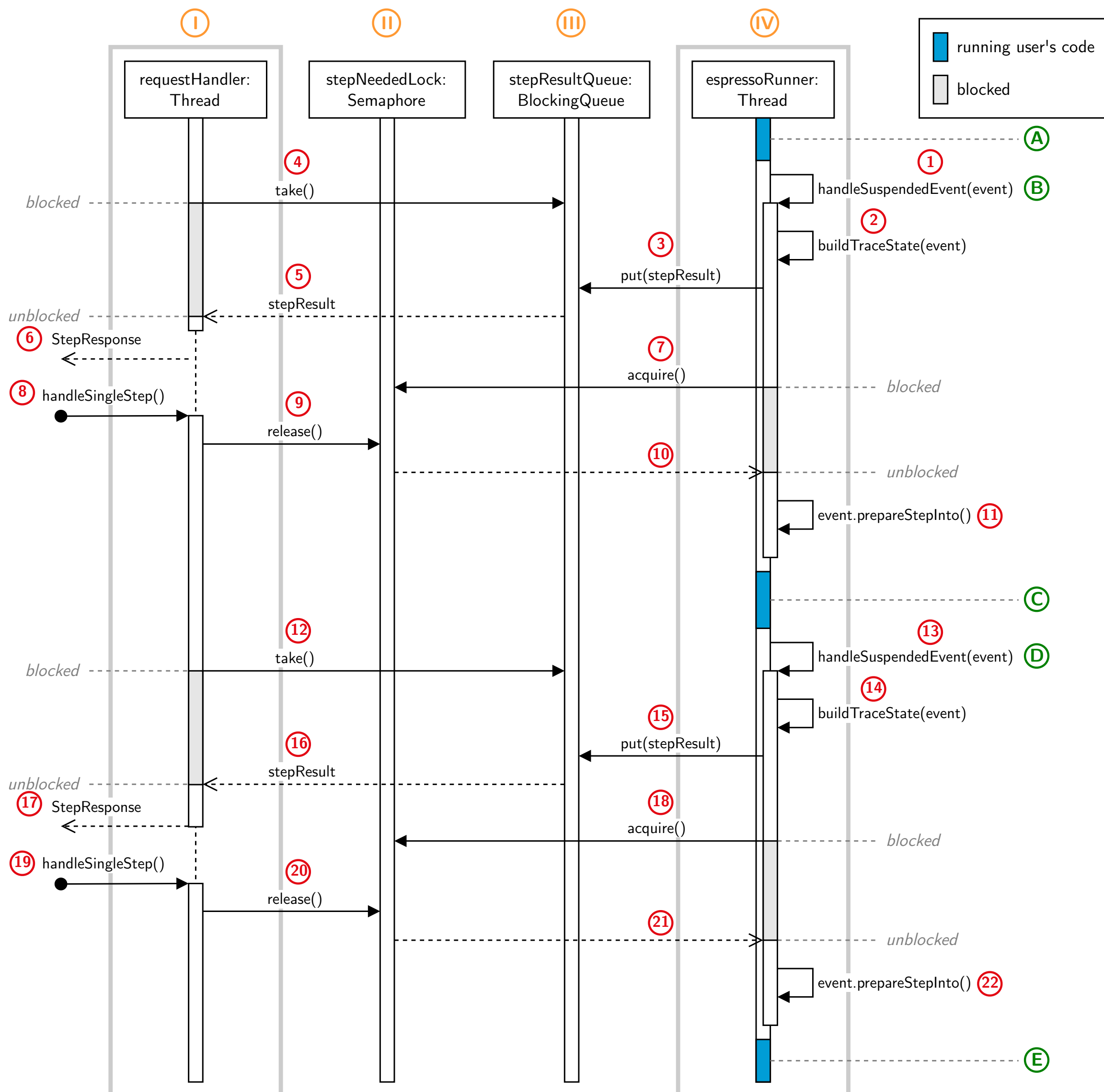


Figure 17: Sequence Diagram of the Debugging Process with Espresso Executing the Example in Figure 16.

- **Heap objects**. Trace states include an array of heap objects. Extraction steps are explained in Section 5.3.1.
- **Loaded classes**. Loaded classes are mainly needed for static fields. Processing loaded classes is covered in detail in Section 5.3.2.
- **Output, error and input**. Trace states contain information about I/O operations. This information is used for the console in the frontend. Section 5.3.3 covers how these are gathered.

- **Input buffer information**. The input buffer information is used to visualize the internal state of the input methods. It contains the following information:
  - Which characters of the input stream have already been processed,
  - which characters are still in the input buffer but are not processed yet,
  - which input function was called last,
  - what the function returned
  - and if the last read was successful.

  The extraction of this information is covered in Section 5.3.4.

5.3.1 Extracting the Call Stack and the Heap

This section is about extracting the state of the execution's call stack and the heap. First, let us take a look what a stack frame from JavaWiz contains:

- **Line Number**. Indicates the line where the stack frame is currently suspended.
- **Class name, method name, method signature, generic signature**. This information is displayed in the frontend and used to determine which method is called within the flowchart component.
- **Local variables**. The main source of information are the local variables. Retrieving them is covered later in this section.
- **Condition values and array access values**. As mentioned in Section 2.1, the source code is modified before running it with a VM. This enables us to trace condition values and array access values. Condition values are used to show which boolean value conditions evaluated to in the desk test. Array access values are used to visualize and animate array accesses in the array visualization. Further details on how the source code is modified and what is saved within the values are explained later in this section.
- **`this`**. If the called method is an instance method, the object the method is called on is stored inside the stack frame.

- **Internal flag**. This flag indicates whether this stack frame should be hidden due to it being within an internal class. Internal classes include classes of the JDK and other classes not of interest.

Most of the above information is quite easy to extract. However, in the following we want to further explain how local variables, heap objects and instrumented values are gathered.

Extracting Local Variables and Heap Objects

This section is about retrieving the local variables within a stack frame and the referenced heap objects. First, we introduce how the Truffle Debug API represents local variables and referenced objects. The current stack frames are accessible through the `SuspendedEvent` inside a `SuspsendedCallback`. Each stack frame has a `DebugScope` which variables can be retrieved with the function `getDeclaredValues` (cf. Figure 18). Each variable is represented as a `DebugValue`. Fields and functions of each object are accessible with the function `get-Properties` (Figure 18 does only show fields). This function allows traversing the referenced heap objects.

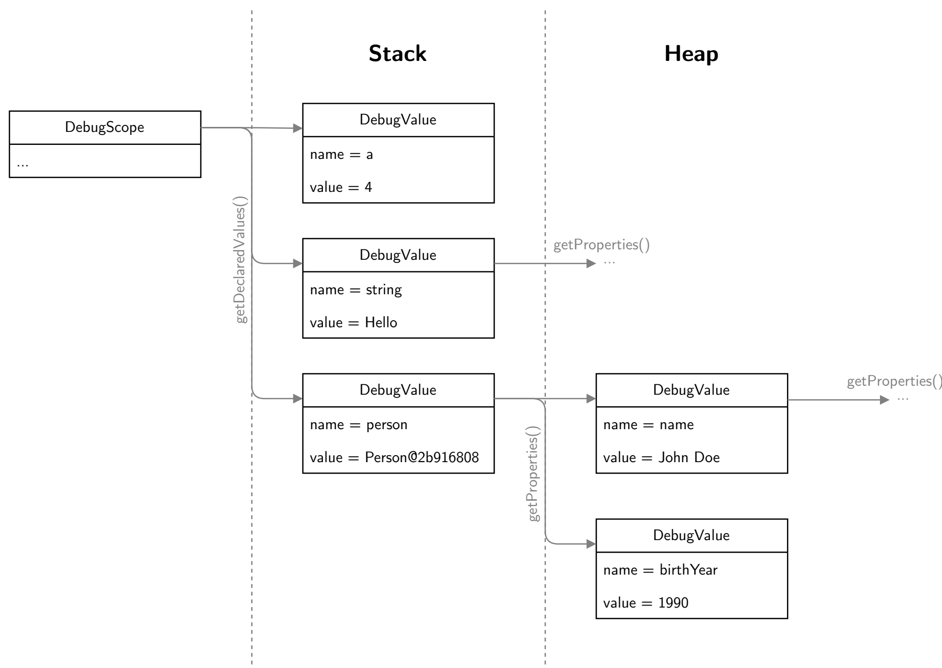


Figure 18: Hierarchy of Debug Values (simplified).

Next, we want to cover how to process this structure in three steps:

1. Loop over all stack frames.

2. Retrieve each open scope.

3. Process the values inside the scope.

Listing 3 computes the local variables for the JavaWiz trace from the stack frames of Espresso. The stack frames are retrieved by using the `SupsendedEvent`. For each stack frame of Espresso, a stack frame for the JavaWiz trace is computed. This works by retrieving the scopes of the stack frame (there are two kinds: `esprDebugScope` from the Truffle Debug API and `esprRawScope` from Espresso), accessing the declared values of the scope and creating the local variables for the JavaWiz stack frame.

```
1  val stackFrames = event.stackFrames
2    .map { esprStackFrame ->
3      val esprDebugScope = esprStackFrame.scope
4      val esprRawScope = esprDebugScope.rawScope as EspressoScope.
         VariablesMapObject
5
6      val localVariables = esprDebugScope.declaredValues
7        .map { esprStackValue ->
8          Var(
9            esprStackValue.name,
10           esprStackValue.getStaticType(esprRawScope),
11           processValue(esprStackValue)
12         )
13       }
14
15     StackFrame(..., localVariables, ...)
16   }
```

Listing 3: Mapping Espresso's Local Variables to JavaWiz Variables (simplified).

Listing 3 uses the function `processValue` for processing the value of local variables. This function is shown in Listing 4. It distinguishes three cases:

- If the value is null, a `NullVal` is returned.
- If the value is a primitive, a `PrimitiveVal` object with the object's string representation is returned.

- If the value is a reference, a `ReferenceVal` object is returned. It contains the ID for the object, which is computed by the function `processHeapItem`.

```
fun processValue(esprValue: DebugValue): Val {
  return if (esprValue.isNull) {
    NullVal()
  } else if (esprValue.isPrimitive) {
    PrimitiveVal(esprValue.toDisplayString())
  } else {
    ReferenceVal(processHeapItem(esprValue))
  }
}
```

Listing 4: Processing Debug Values.

Listing 5 shows the function `processHeapItem` which was used in Listing 4. First, the ID of the object is retrieved. If the object has been processed already, the ID simply is returned. Otherwise, the following cases are distinguished:

- If the value is a string, it gets further processed with the function `processString`. This function creates an `HeapString` object for the JavaWiz trace. The string's byte array is processed with the function `processArray`.

- If the value is an array, the object is processed with the function `processArray`. Within it, all elements are processed and a `HeapArray` object is created for the JavaWiz trace.

- Otherwise, the object is further processed with the function `processArbitraryObject`.

```
fun processHeapItem(esprValue: DebugValue): Long {
  val esprId = esprValue.getHeapId()
  if (esprId !in processedHeapIds) {
    processedHeapIds += esprId
    if (esprValue.isString) {
      processString(esprId, esprValue)
    } else if (esprValue.isArray) {
      processArray(esprId, esprValue)
    } else {
      processArbitraryObject(esprId, esprValue)
```

```
    }
  }
  return esprId
}
```

Listing 5: Processing Heap Objects.

The function `processArbitraryObject` used in Listing 5 is shown in Listing 6. First, the internal representation of the object's class is retrieved. Then, the function distinguishes two cases to retrieve the object's fields:

- If the class is an internal class, its fields are not traced, i.e., the list of fields is empty. This check is performed by matching the class' name against internal class patterns.
- Otherwise, the field table of the class is accessed. Each field is mapped to a JavaWiz variable. Their values are processed recursively with the function `processValue` which has been covered earlier.

```
fun processArbitraryObject(esprId: Long, esprValue: DebugValue) {
  val klass = EspressoInterop.getInteropKlass(esprValue.rawValue
     as StaticObject)

  val fields = if (outerClassMatchesOuterClassPattern(
    klass.nameAsString,
    internalClassPatterns)
  )
    emptyList()
  else klass.fieldTable
    .map {
      Var(
        it.nameAsString,
        toReadableType(it.typeAsString),
        processValue(esprValue.getProperty(it.nameAsString))
      )
    }

  heapItems += HeapObject(esprId, esprValue.getType(), false,
     fields)
```

```
19 }
```

Listing 6: Processing Arbitrary Heap Objects (simplified).

Note, that Espresso only exposes public fields through the Truffle Debug API. Therefore, all fields are set to be declared publicly while parsing. Since the unmodified source code is compiled as well, access violations are appropriately reported to the user.

This completes the extraction of the call stack and the referenced heap objects. To summarize, we loop over each stack frame, process its local variables and process the heap objects referenced by them. Subsequent references are processed in a recursive manner.

Collecting Instrumentation Values

This section covers how the instrumentation values are collected. First, we want to introduce how the source code instrumentation works (cf. Figure 19). The unmodified source code is at the top and the modified one is at the bottom. There are three modifications done in the example, which are highlighted in gray. There is one modification for a condition and two modifications for array accesses. Here is how they are modified:

- Each condition is replaced with a call to a JavaWiz instrumentation method. The parameters contain the condition itself and a condition ID unique for this source file.
- For array accesses, only the index value is instrumented by a method. The parameters contain the accessed array, the index value, an access ID unique for this source file and the accessed dimension.

Since each expression is still only executed once, this form of instrumentation is safe and does not cause any side effects.

Listing 7 shows the JavaWiz debug class used for intrumentation. It contains the two functions used in Figure 19:

- The function `recordCondition` returns the condition's value.
- The function `recordArrayAccess` returns the accessed index.

Figure 19: Source Code Instrumentation with JavaWiz.

```
1  package jwdebug;
2  public class $JavaWiz {
3    public static boolean recordCondition(boolean value, int
        conditionId) {
4      return value;
5    }
6
7    public static int recordArrayAccess(Object array, int index, int
        arrayAccessId, int dimension) {
8      return index;
9    }
10 }
```

Listing 7: The JavaWiz Instrumentation Class.

How can these instrumentation functions be used to collect the instrumented values? When the VM is suspended in one of the instrumentation functions, the relevant information can be obtained by accessing the parameters. After retrieving the values, the VM is resumed.

Listing 8 shows how this is implemented with the Truffle Debug API. The suspended event contains the source section it is currently suspended in. With its path, we can determine if Espresso is suspended inside the JavaWiz debug class. After tracing the instrumented values, the VM is resumed, thus tracing the instrumentation values does not yield a trace state.

```
1 if (event.sourceSection.source.path == JAVAWIZ_PATH) {
2   handleInstrumentationTracing(event)
3   vm.resume()
4 } else {
5   handleNormalTracing(event)
6 }
```

Listing 8: Differentiating between Instrumentation Tracing and Normal Tracing (simplified).

Finally, inside the function `handleInstrumentationTracing`, the parameters of instrumentation functions can be captured. This is done similarly to how local variables are processed. Together with some other information, these are collected for the next trace state.

5.3.2 Extracting Loaded Classes and Statics

This section covers retrieving loaded classes and their static fields. Espresso does not support querying loaded classes nor are there class load events as with the JDI. With these limitations, here are the steps to retrieve loaded classes and their statics with Espresso:

1. Determine relevant class names in the user's code.

2. Lookup these classes with the class loader of Espresso.

3. If they have been initialized, their static fields can be accessed.

Listing 9 shows how this procedure is implemented. Both the class names as well as the class loader are accessible through the Espresso runner. The class names are then mapped to their language agnostic class representation. Together with the Espresso internal class representation, they are mapped to JavaWiz' data structure for loaded classes. For their static fields, the field table is mapped to variables of JavaWiz. Each field's value is processed similarly to the value of instance fields in function `processValue`.

```kotlin
fun processLoadedClasses(): List<LoadedClass> {
  val classNames = espressoRunner.classNames
  val esprClassLoader = espressoRunner.context.getBindings("java")

  return classNames
    .mapNotNull { esprClassLoader.getMember(it) }
    .map { classValue ->
      val klass = classValue.`as`(ObjectKlass::class.java)
      LoadedClass(
        klass.nameAsString,
        klass.staticFieldTable.map { field ->
          Var(
            field.nameAsString,
            toReadableType(field.typeAsString),
            processValue(classValue.getMember(field.nameAsString))
          )
        }
      )
    }
}
```

Listing 9: Processing Loaded Classes (simplified).

Note, that only public static fields are accessible through the Truffle Debug API currently. To overcome this, static fields are changed to be declared publicly while parsing, similarly to how it is done with non-static fields (cf. Section 5.3.1).

5.3.3 Standard I/O Stream Handling

This section is about handling standard I/O streams with Espresso. Whenever a trace state is generated, the output streams are read and reset. The read string is then included in the trace state.

Handling the standard input stream is a bit more challenging than that. Figure 20 illustrates how this is accomplished. Whenever the function `read` of the standard input stream is called (1), the native call is substituted with the call to a custom input stream. This input stream stores the user input as a string. As there is no input initially, the frontend is notified and the execution is blocked (2). The request for new input is sent via the WebSocket connection

(3). When the user inputs a new string (4), it is sent back via the WebSocket connection. Upon receiving the input, the custom input stream simulates a regular input stream. This is done by returning byte for byte of the input on consecutive calls of the function `read` (6).

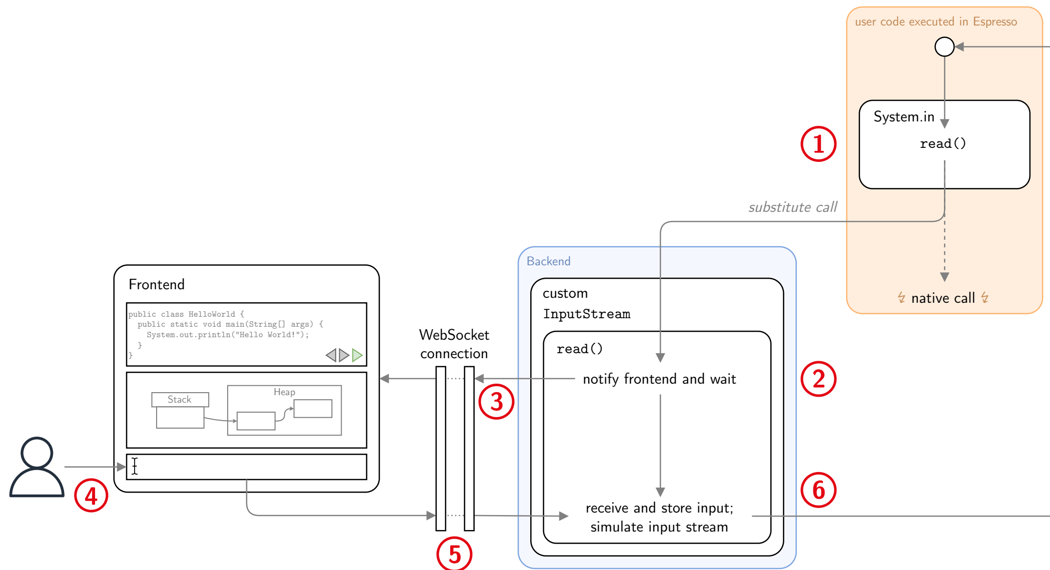


Figure 20: Handling Input via the Standard Input Stream in Espresso.

5.3.4 Extracting Input Buffer Information

Finally, this section is about extracting input buffer information. The utility class `In` is distributed together with JavaWiz. It is used in the programming courses of the SSW. The class offers functions to read various forms of input from the standard input stream or a file. To aid student's understanding how this class processes input, a visualization for its internal data structures was developed by Melissa Sen [15].

The class works by reading and processing input character by character. To check if an operation has concluded correctly, the class offers a function `done`.

By visualizing the input buffer, students can grasp what the class has already read and processed (cf. Figure 21). The visualization of the buffer is split into two parts: (1) what has been read (in gray) and (2) what is left inside the buffer (in green). These parts are called *past* and *future* respectively. Additionally, the visualization shows which function was called last, which value it returned and the current return value of the function `done`.

All this information is accessible with public functions of the class. Hence, to extract the information, these functions are called while building a trace state.

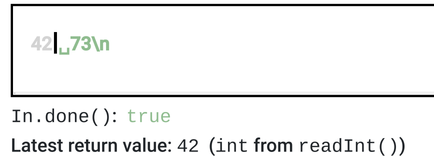


Figure 21: Screenshot of the Input Buffer Visualization.

Next, we want to take a look at how this information can be extracted when debugging a program using the input utility class `In` in Espresso, which is shown in Listing 10. A reference to the class `In` is stored in `inClass`. If the class has not been stored yet, the class is looked up first. For that, the class names found in the user's code are examined. If the user has not supplied the class, JavaWiz offers a built-in one which is used instead. Next, the class representation is queried with the class loader. After checking if the class has been initialized, it can be used for extraction. If the class does not contain all expected functions, a failure is returned. Otherwise, each function is queried and executed. Finally, the function's return values are converted to host objects and returned in an `InputBufferInformation`.

```
1  fun processInClass(): InputBufferInfo {
2    if (inClass == null) {
3      val inClassName = espressoRunner.classNames.find { it.split(".").last() == IN_CLASS_NAME }
4        ?: IN_CLASS_NAME
5      val inClassValue = espressoRunner.context.getBindings("java").getMember(inClassName)
6        ?: return InputBufferInfo.EMPTY
7      if (inClassValue.`as`(ObjectKlass::class.java).state >= ObjectKlass.INITIALIZED)
8        inClass = inClassValue
9      else
10       return InputBufferInfo.EMPTY
11   }
12
13   if (!inClass.memberKeys.containsAll(EXPECTED_IN_KLASS_MEMBERS)) {
14     return InputBufferInfo.FAILED
15   }
16
17   return InputBufferInfo(
```

```
18      inClass.getMember(IN_PAST_SIG).execute().asString(),
19      inClass.getMember(IN_FUTURE_SIG).execute().asString(),
20      inClass.getMember(IN_DONE_SIG).execute().asBoolean(),
21      inClass.getMember(IN_LATEST_VALUE_SIG).execute().asString(),
22      inClass.getMember(IN_LATEST_METHOD_SIGE).execute().asString()
23    )
24  }
```

Listing 10: Extracting Input Buffer Information (simplified).

Chapter 6

JavaScript Backend

Replacing the VM with Espresso enables the second goal of this thesis: compiling the backend to JavaScript using WebImage. Utilizing Espresso, the backend does not depend on a VM in a separate process anymore. As a result, it is possible to compile the backend to JavaScript. Executing the backend with JavaScript has some challenges, which will be covered in this chapter:

1. Architectural changes resulting by executing the backend in a JavaScript runtime are covered in Section 6.1. These modifications include adapting the communication and synchronization methods used between frontend and backend.

2. A JavaScript runtime does not feature native access operations. This leads to the need of substituting native file system calls. These replacements are described in Section 6.2.

3. The JavaWiz backend uses reflection. However, WebImage cannot detect every reflection case automatically and is not able to support used dynamic reflection [14]. Configuring reflection is documented in Section 6.3.

6.1 Architectural Changes

This section is about the architectural changes needed to run the backend in JavaScript. With the current version of JavaWiz, Java code is run in a separate process. First, JavaScript does not allow creating processes. Additionally, it does not allow to directly run Java. Because of

this, Espresso is used to run Java bytecode. This way, the user's code can directly run within the backend. Yet, we still need a way to run the backend concurrently with the frontend. JavaScript has a solution for that: *Web Workers* [29].

Web Workers allow executing JavaScript code in a separate thread from the main thread [29]. A Web Worker is started by providing a JavaScript file. Since the output of WebImage is a JavaScript file, it can be executed directly by a Web Worker. Once started, a Web Worker provides an interface for communication similar to WebSockets. It allows reacting to messages and posting messages to other listening threads. Utilizing Web Workers, the backend can run simultaneously to the frontend. There are two alternatives to achieve this:

1. Follow the architecture of the Java version running in two threads, i.e., spawn two Web Workers, one for the backend logic and one for the VM. There are two ways to accomplish this:

    - Create separate scripts, one for the backend logic and one for Espresso.

    - Use the same script yet differentiate what to perform when reacting to requests.

2. Adapt the architecture of the Java version and let the backend run in a single thread. As previously mentioned in Section 2.1 and Section 5.2, the backend thread is responsible to decide when to send the collected trace states to the frontend. Concurrently, the other thread is running Espresso. If the whole backend is run in a single thread, both the backend logic and Espresso have to be executed in the same thread.

Both of these solutions have their pros and cons, but we have decided to implement the second one and run the backend in a single thread. Adding the logic of determining when to send the collected trace states to the Espresso debugging process is less complex.

Figure 22 shows how the frontend and the backend communicate when operating in JavaScript. The frontend is executed by the main thread on the left. Concurrently, the backend is run inside a Web Worker on the right. Compile requests as well as responses from the backend are sent through the Web Worker interface.

When running the backend with JavaScript, it should behave the same way as when run with Java. Hence, there needs to be a way to block Espresso. In other words, the Web Worker executing Espresso needs to be blocked. Blocking a Web Worker is possible with

JavaScript's Atomics [33]. They provide the methods wait and notify which are modelled on Linux futexes [33, 34]. With them, it is possible to interrupt and wake up Web Workers. Atomics work on instances of SharedArrayBuffer and ArrayBuffer. Since the buffer is used in multiple threads, a SharedArrayBuffer is needed. By making use of atomics, Espresso can block itself from further execution and later can be woken up by another thread to continue execution. Thus, a SharedArrayBuffer is included in Figure 22 for synchronization purposes.

When creating an integer array that views the SharedArrayBuffer, the buffer can be accessed with Atomics [33]. Listing 11 shows an example for this. First, a SharedArrayBuffer is created. Then an integer array is created that views the buffer. The function wait receives the array, an index and a value. If the SharedArrayBuffer contains the given value (1) at the given index (0), the calling thread is blocked. The function notify receives the array, an index and the amount of threads which should be notified. When called, the given number of threads (0 means all of them) waiting for that index is unblocked.

```
1 const sab = new SharedArrayBuffer(1024);
2 const int32 = new Int32Array(sab);
3 Atomics.wait(int32, 0, 1);
4 Atomics.notify(int32, 0, 0);
```

Listing 11: Examples for the Use of Atomics [33].

The SharedArrayBuffer is not only used for synchronization purposes but also for data transfer. Step requests and user input are sent through the buffer (cf. Figure 22).

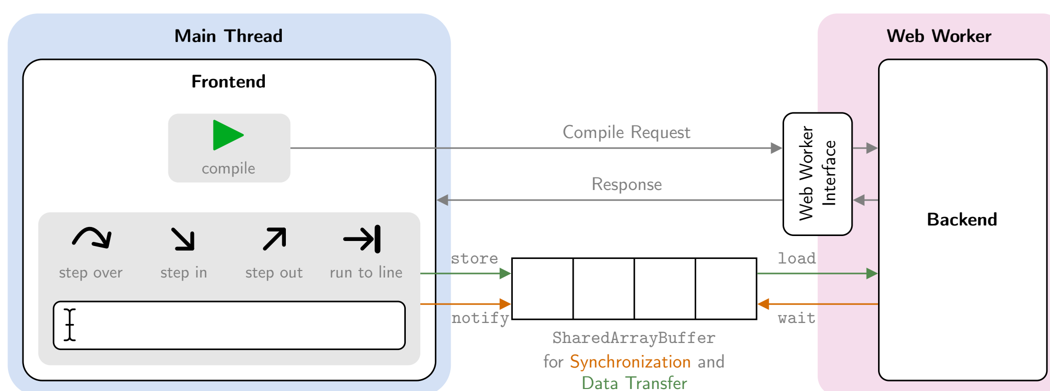


Figure 22: Communication Architecture of the JavaWiz Backend in JavaScript.

So far, we have decided to use a single thread for running the whole backend. Additionally, we have established that a shared array buffer has to be used for communication and synchronization purposes. There are two open issues:

1. How are the requests sent through the shared array buffer?

2. How does the debug process of Espresso look like with the implemented changes?

These issues will be discussed in the following two sections.

6.1.1 Sending Requests through a Shared Array Buffer

To send requests through a shared array buffer, we need to define a protocol. This protocol has to work with integers only, since the relevant functions of `Atomics` only work with integer arrays [33]. Before defining such a protocol, we need to enumerate what requests can contain:

- **Task kind**. The buffer should contain which task is requested. Tasks include *step-into*, *step-over*, *step-out*, *run-to-line*, *compile* and *input*. The content of compile requests does not need to be sent over the buffer, since compile requests are handled in separate events.

- **Reference stack depth**. Both step-over and step-out tasks include a reference stack depth. With this depth, it can be determined if the debugger has run far enough.

- **Line number**. The run-to-line task specifies the line until which the program should continue.

- **Strings**. Run-to-line tasks contain the class name the line is in. Furthermore, console input contains the entered text as a string.

Using this information, we can define a protocol to store requests and input in the buffer (cf. Figure 23). The first element contains the task ID to identify the task kind (step-in, step-out, input, ...). Depending on the task, the second element holds the reference stack depth or a line number. If a string is sent, the third element contains the length of it. Starting from element 4, the string is stored.

Using the proposed protocol requires a buffer that can store at least four integers. The greater the buffer size, the greater the capacity for transmitting strings. If a string is longer than the buffer, it is sent in fragments.

Figure 24 shows the buffer's content with two examples. Gray cells may contain arbitrary values. A step-out request is stored in the upper buffer. The first index contains the corre-

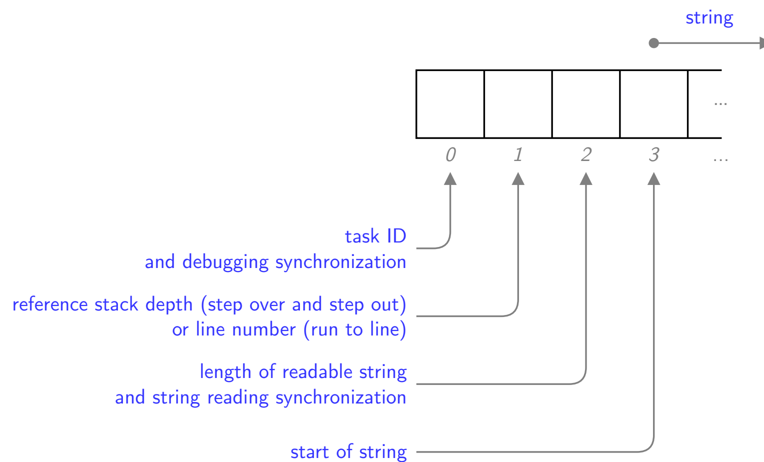


Figure 23: Index Assignment for a Shared Array Buffer Used for Communication.

sponding task ID (`3`) and the second one contains the reference stack depth (`4`). With this request, the program would continue until the stack depth is ≤ 4. The second example shows a buffer containing user input, indicated by the task ID `7`. The length of the sent input (`5`) is stored at the third element. Staring at the fourth element, the input string (`Hello`) is encoded as integers.

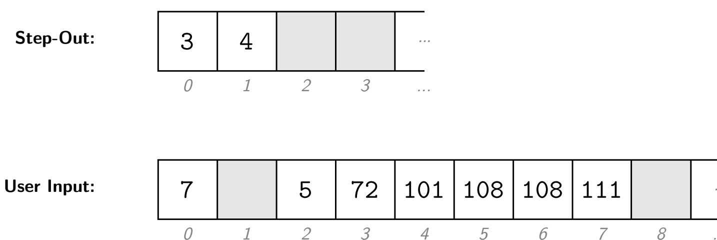


Figure 24: A Step-Over Request and a User Input Stored Inside a Shared Array Buffer.

6.1.2 Debugging Espresso after its Compilation with WebImage

This section covers the debugging process with Espresso after compiling it with WebImage. Figure 25 depicts a sequence diagram illustrating the workflow. The general structure is similar to debugging Espresso (cf. Figure 17 of Section 5.2). Note, that the diagram has been simplified. Methods using the shared array buffer are shown as instance methods, but rather are static methods of `Atomics`. The buffer is passed to these methods alongside the given arguments. Note that the function `wait` is blocking if the passed value is stored at the given index. More precisely, it does not wait until the passed value is stored there.

There are four lifelines to consider in Figure 25:

I — The frontend is responsible for user interactions and for visualizing the program state. Initially, the *user interface (UI)* is blocked since there is an ongoing request pending, in this case a step into.

II — As discussed earlier, a shared array buffer is used for synchronization and communication between the two threads. For this example, only the first index is relevant. Initially, the buffer stores the task ID for step into (1) in this index, as this was the last unfinished request by the frontend.

III — Espresso is executed in the backend, which executes the user's code and handles debugging. Starting the sequence, the backend is executing user code.

IV — The backend creates a response builder for each request. Response builders decide when the corresponding request has been fulfilled and thus determine if a response is sent to the frontend. Therefore, response builders replace the second thread in the backend. Initially, a response builder is processing the last unfinished request, a step-into task.

After executing user code, Espresso encounters a suspension point and triggers the execution of a suspended callback. While handling the suspended event (1), a trace state is built (2) and added to the response builder (3). Subsequently, the builder adds this event to its collected trace states (4) and then checks if the response should be sent to the frontend (5). To determine this, it tries to build a response with the current collected information (6). Within this function, a check is performed if the collected information fulfills the request. For example, with the step-out task in Figure 24, the backend would check if the current stack depth is less than or equal to 4. If so, a response is built. Since the current request is a step into, a single trace state fulfills the request and a response is built. A new request is required before continuing execution. Consequently, the buffer is prepared for blocking (7), by storing 0 for the task ID. This indicates that the execution should be halted. Afterward, the step result is sent to the frontend (8). The frontend triggers a redraw of the visualizations (9) and unblocks the UI. The order of operations is important in this situation: If the result is sent to the frontend (8) before preparing the buffer (7), user input may cause a new request. This request would be stored inside the buffer and could be overwritten by the response builder.

After sending the result, the response builder is not needed anymore (10). Since the task ID inside the buffer is still 0, the backend blocks itself (11). Meanwhile, the user requests a step into (12). Hence, the corresponding task ID is stored inside the buffer (13). All

waiting threads are notified (14), which leads to the backend being unblocked again (15). Subsequently, the backend reads the new task in the buffer (16) and creates a new response builder (17). Finally, it prepares a step into (18), since we need to record every step of the program, as discussed in Section 5.2. At this point, we have reached a similar state as the initial one of the diagram. The process can start over again.

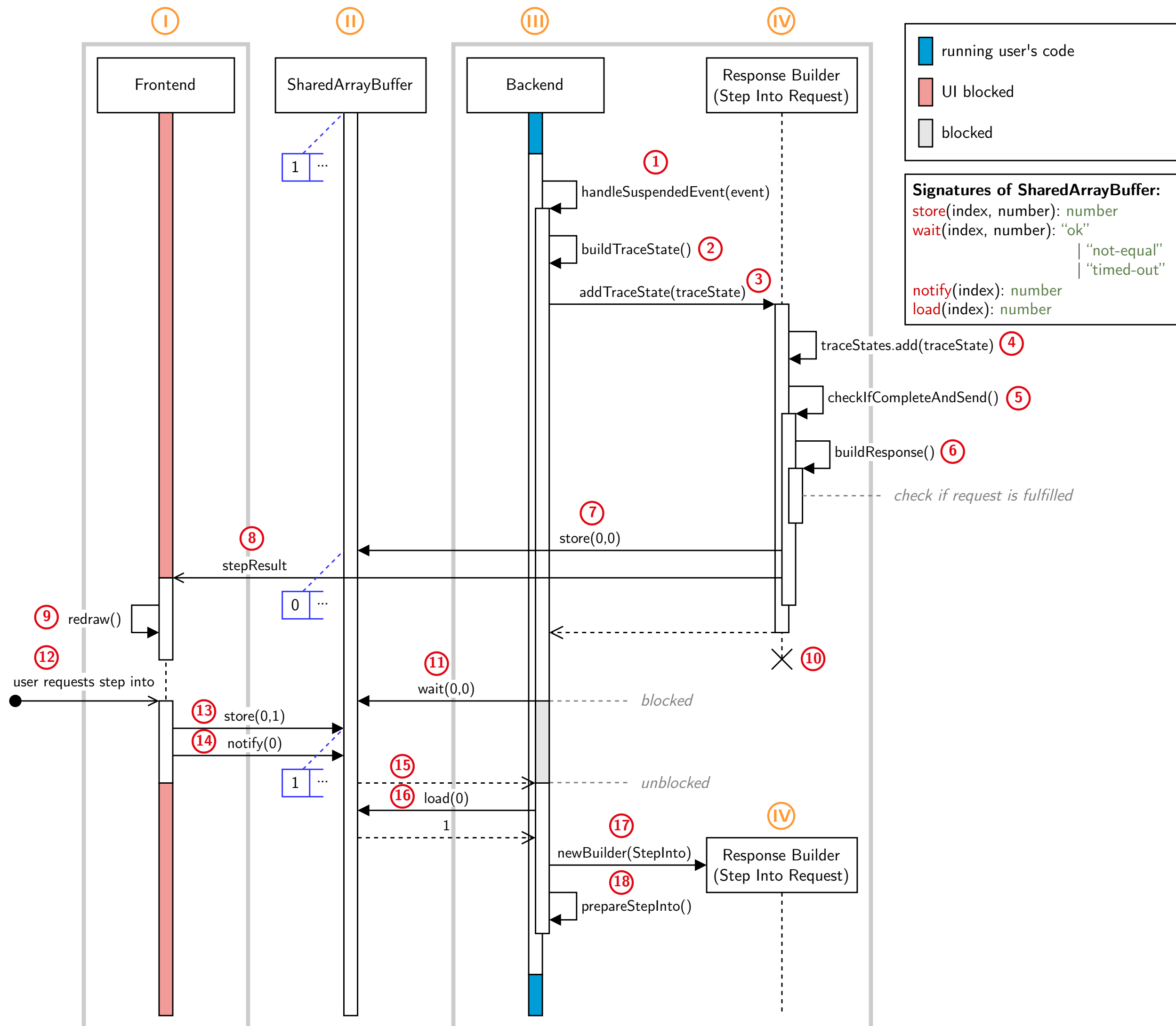


Figure 25: Sequence Diagram of the Debugging Process with Espresso After Compiling it with WebImage (simplified).

6.2 Substituting File I/O

This section is about substituting native file system calls. Normally, Espresso has access to the file system through the host process. Since JavaScript does not have access to a file system, native file calls have to be handled differently.

Since JavaScript has no access to the user's file system, WebImage employs an in-memory file system. Compiled programs can thus still make use of a file system. Figure 26 illustrates which parts of JavaWiz' architecture have access to the in-memory file system. The backend, the compiler and Espresso all have access to it. This allows for storing source files and generated class files inside the in-memory file system. While Espresso itself works with the file system, native calls by code executed within Espresso are not rerouted automatically.

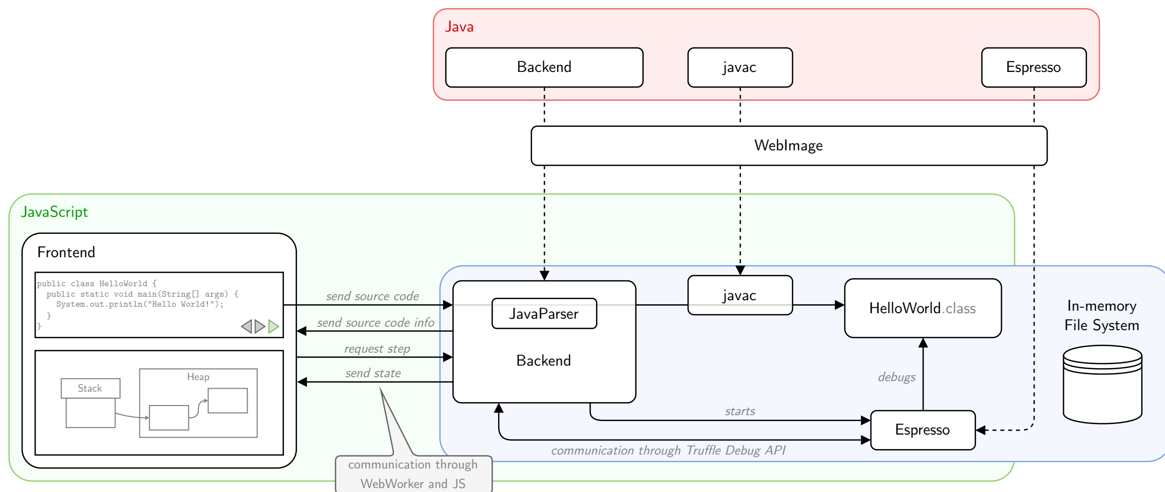


Figure 26: The Architecture of JavaWiz With Espresso as a VM, Compiling it With WebImage and an In-Memory File System.

Figure 27 contains an exemplary file tree of the in-memory file system. There are directories for source files, for class files and for the JDK. The working directory is `work`.

```
classPath
└── HelloWorld.class
jdk
└── modules
    └── java.base
        └── ...
sourcePath
└── HelloWorld.java
work
└── input.txt
```

Figure 27: Exemplary File Tree for the In-Memory File System.

To provide file I/O for user's code, native file I/O calls are substituted (cf. Figure 28). For example, the function `read` of `FileInputStream` and the function `write` of `FileOutputStream` are substituted. With the substitutions, the file streams are rerouted to the in-memory file system. Return values are forwarded to the caller. These substitutions have to be built into Espresso itself. Consequently, a special build of Espresso is needed to compile JavaWiz with WebImage.

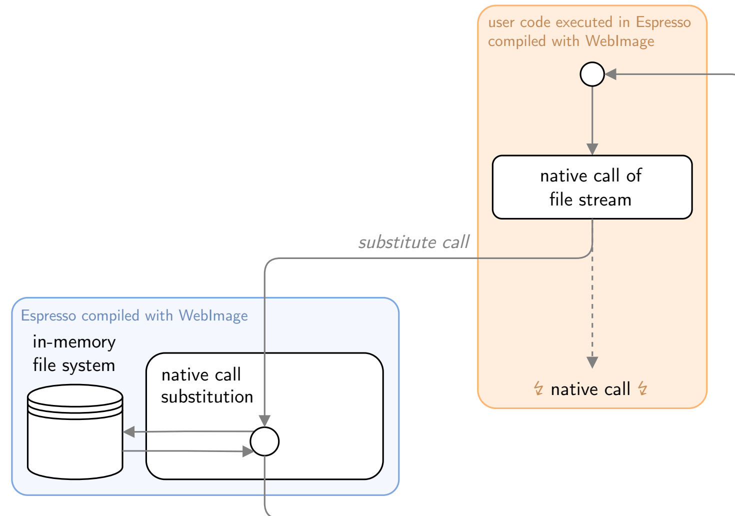


Figure 28: Substituting I/O Calls After Compilation With WebImage.

6.3 Supporting Reflection

WebImage tries to detect usages of reflection and supports such usages when executing the generated JavaScript code [14]. It is not possible for WebImage to take reflection scenarios into account that are unknown during build time. To enable dynamic reflection with WebImage, a configuration may be passed to the build process [9]. This configuration has to contain all relevant usages of reflection.

To recap Section 4.2, the JavaWiz backend uses reflection to accomplish the following:

1. For the communication between the frontend and backend, JSON is employed. Reflection is needed to parse received JSON objects and to generate JSON objects.

2. As mentioned in Section 2.1, the user's code is parsed and modified. For this, the third-party parser JavaParser [16] is used. This parser uses reflection on internal objects to provide access to the AST.

To compile JavaWiz with WebImage, a configuration of reflection usages is needed. This is straightforward for case 1, creating and parsing JSON objects. In contrast, the reflection used by the JavaParser is harder to identify. Thus, creating a configuration by hand seems to be impractical.

WebImage provides a solution for this: a tracing agent. This agent creates, among others, a configuration for reflection (cf. Figure 29). It gets attached when a JVM is started.

During executing Java bytecode, the tracing agent keeps track of which classes are accessed through reflection. The generated configuration can then be passed to the building process of WebImage.

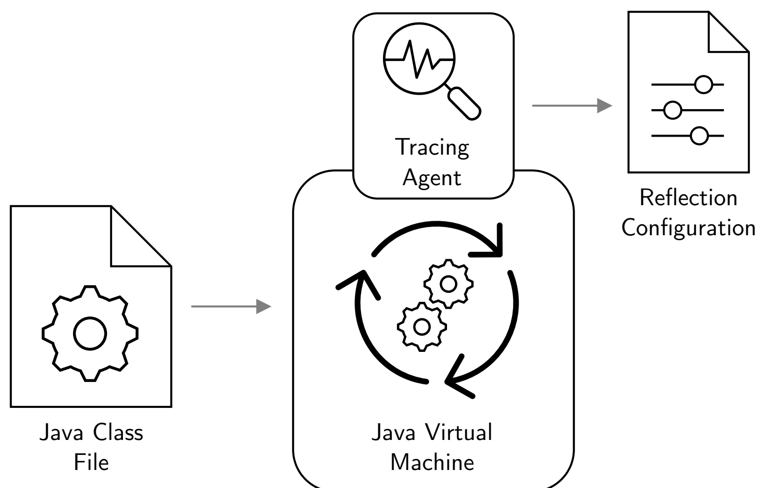


Figure 29: Automatically Generating a Configuration for Reflection.

To generate a configuration for reflection, we use unit tests delivered with the JavaParser. A configuration is created for each test. The collected configurations are then merged with one needed for creating and parsing JSON objects.

Note, that there are plans to eliminate the dependency on JavaParser. If the JavaParser was not used anymore, the configuration for reflections would only need to contain cases for communication purposes.

Chapter 7

Handling Multiple Files

The third goal of this thesis is to add support to handle multiple files. Currently, the web version of JavaWiz is only able to handle a single source file. First, let us revisit the requirements of this goal, which were already discussed in Section 4.3:

1. It should be possible to edit multiple source files in the frontend.

2. Additionally to source files, the frontend should also support creating and editing text files. Text files should stay updated during execution.

3. If a new file is created during execution, it should be exposed to the user in the frontend. It does not need to be editable.

We will cover the implementation of these requirements separately for the frontend (Section 7.1) and the backend (Section 7.2). Before that, we briefly explain how they interact. When the user starts the program, all files are sent to the backend. Further, whenever a file is modified during execution, the frontend is notified. We decided that file changes in the frontend should not be propagated to the backend during execution. The primary reason behind this decision is that it would not be apparent to the user when these modifications are exposed to the backend. While Espresso is running, it is not possible to update files in the file system as the backend is busy. Therefore, we decided that file changes cancel the execution.

7.1 Extending the Frontend

Let us begin by defining what needs to be added to the frontend:

- There needs to be a tab-like view to switch between files.
- The frontend should react to file related notifications accordingly. Either it updates the corresponding file or it creates a new one.

7.1.1 Storing Multiple Files in the Frontend

To understand how the frontend can store multiple files, we first take a look at how JavaWiz currently handles its single source file (cf. Figure 30). As stated in Chapter 1, the frontend is using the JavaScript framework Vue.js [5]. The main view `Home.vue` contains all components needed for the frontend. One of the components is `TheCodeEditor.vue`, which utilizes the *Monaco Editor* [35]. It stores the entered text as a value. This value is bound to the `modelValue` of the component which is kept in sync with the property `editorText` in `Home.vue`. When the user requests to run the code, the editor text is sent to the backend.

Why is the source code not solely stored within the component, but also inside `Home.vue`? `Home.vue` is acting as conductor between all the components. Since the editor's text is needed by other components, it is stored globally.

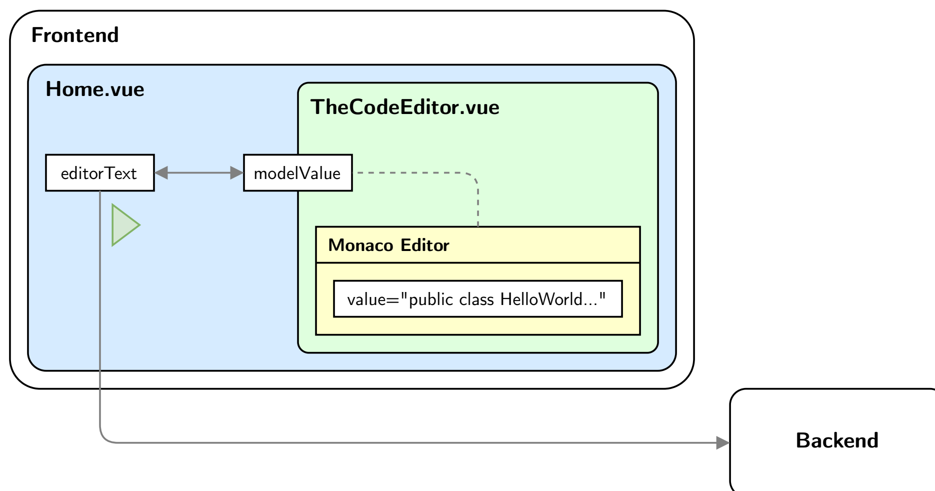


Figure 30: Old Handling of Source Code in the Frontend (simplified, inspired by [1]).

Figure 31 shows how the frontend can be extended to handle multiple files. The implementation is heavily inspired by the Monaco Editor's playground's source code [36]. The following objectives are pursued:

- Use only a single editor instance to save resources.
- Create a new model for each file. Models are used by the Monaco editor to store text, an undo stack and more.
- When switching between files, change to the corresponding model in the editor. Save the view state (cursor position, scroll position, undo history, ...) of the previous model and restore the one of the now viewed model.

Figure 31 illustrates the approach with two open files. The Monaco Editor stores a model for each file. Each model contains its text as a value and a reference to its *Uniform Resource Identifier (URI)*.

Instead of the property `editorText` `Home.vue` contains a file manager. The file manager keeps track of all files, each linked to its `FileData`. The file data stores the URI of the file, whether the file is read only and other properties not depicted in the diagram.

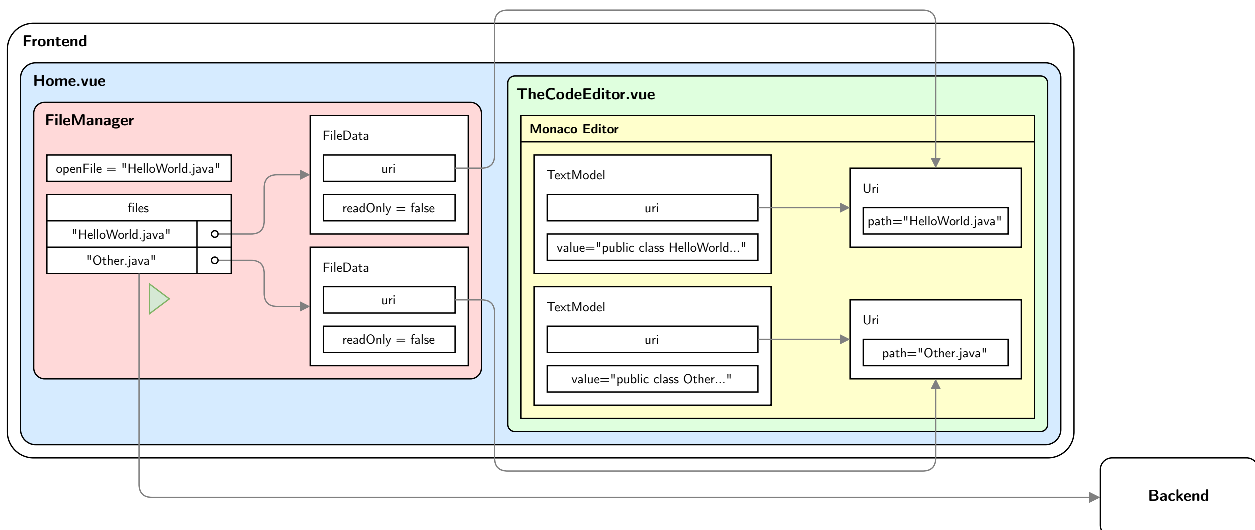


Figure 31: New Handling of Source Code in the Frontend (simplified).

Whenever the content of a file is needed, its URI is used to look up the corresponding model in the editor. The file's content is then accessible through that model.

7.1.2 Reacting to File Changes

After explaining how multiple files are stored in the frontend, we want to cover how to react to file changes. For that, we have to distinguish between a file modification and the creation of a file:

1. If the file already exists within the frontend, we have to look up the corresponding model and change its contents.

2. If a new file has been created, a model with the corresponding URI has to be created. The property `readOnly` within the file data has to be set to `true`, as changes are not sent to the backend.

7.2 Adapting the Backend

Let us recap what has to be adapted in the backend to support multiple files:

- Distinguish between source files and text files, i.e., store source files at the source path and text files in the working directory. This way, a user can access the text files via their relative path.

- Monitor all opened file output streams and notify the frontend about changes within files.

This approach was already presented in Figure 14. With the information from Section 6.2, we can specify it more precisely (cf. Figure 32). When the user starts the execution, the files are sent to the backend. There, the files are split up into source files and text files. Source files are stored at the source path, while text files are inserted into the working directory. Subsequently, the source files are parsed, modified and compiled. Then, Espresso is launched and keeps track of all opened file streams. With the help of Espresso, the backend checks for modifications or creations of files and the frontend gets notified if necessary.

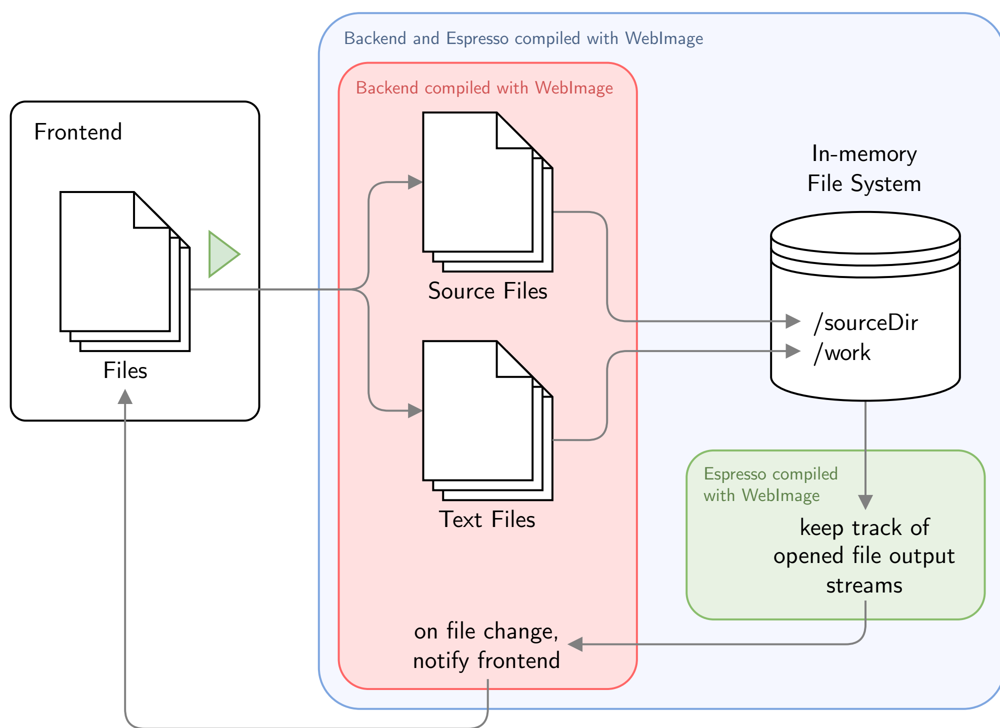


Figure 32: More Detailed Approach for Handling Multiple Files in the Backend.

Chapter 8

Loading Predefined Examples

As pointed out in Chapter 1, providing JavaWiz via a website mitigates the complexity for novice users. To recap, it eliminates the need to download VS Code, install a plugin and set up an appropriate JDK. Yet, an additional advantage has been left out: The URL of JavaWiz can be used to embed additional information. In particular, we could provide information to load JavaWiz with a specific predefined setting. As a part of this thesis, we developed a technique to load the code of a predefined example into JavaWiz. This chapter covers how this works and demonstrates a website with predefined examples.

8.1 Loading a Predefined Configuration

Before explaining how this was implemented, we want to set the requirements of this feature:

- It should be possible to load examples by opening a predefined URL.
- Adding new examples should be independent of building the frontend. In other words, the frontend should be able to load examples without them being known at build-time.
- Links to examples should look like this: `<URL of JavaWiz>/<Example Name>`.

First, one needs to define how examples are stored on a server and how they are identified. This is accomplished as follows:

- Each example is stored within a unique directory.

- Each directory contains Java source files and text files, possibly in subdirectories.
- Examples are identified by the name of the directory they are stored in. The name of the directory should be the `<Example Name>` for the URL.
- The examples are stored on the same server as JavaWiz in a directory named `examples`. Figure 33 shows an exemplary directory structure of such a server. Every file and directory but the directory `examples` is output when building the JavaWiz frontend.

```
backend.js
backend.js.bin
css
└── ...
editor.worker.js
editor.worker.js.map
examples
├── GCD
│   └── ...
├── HelloWorld
│   └── ...
├── IntegerList
│   └── ...
├── MartrixMultiplication
│   └── ...
└── ReadInt
    └── ...
favicon.svg
fonts
└── ...
img
└── ...
index.html
js
└── ...
```

Figure 33: Exemplary Directory Structure of a Server Hosting JavaWiz and Examples.

After defining how examples are stored and identified, we will take a look at how they can be loaded. In the frontend, it is possible to retrieve a file from a server using the Fetch API [37]. Yet, we run into a problem here: One can only fetch from a web server if the file name is known. Also, it is not possible to fetch a directory. To solve this, the JavaWiz frontend needs to know which files it has to fetch. There are different options to achieve this:

1. The server could provide a REST API [38] to get all files associated with an example.
2. Whenever a directory for an example is requested, the server could be configured to return an index HTML file.

3. Alternatively, a PHP [39] script could be executed on the server to obtain a list of files.

4. A configuration file with a predefined name could be stored within each directory. It is written by the example author and contains all file names for an example.

We decided to use the last approach, since it has two advantages:

1. With the configuration, the example author can predefine in which order the files should be opened.

2. The configuration can be extended to hold additional information, such as initially opened visualizations.

Figure 34 shows a more detailed example file structure of the directory `examples` (cf. Figure 33). Each example has its own directory. Each folder contains a configuration file `config.json` written in JSON. Next to the configuration, each directory also contains sources for the example. The example *ReadInt* additionally contains a text file. Whenever JavaWiz should be loaded with an example, the frontend fetches the configuration file. Then, it extracts all relative paths for needed files and fetches them individually. Note, that there could be subdirectories.

```
GCD
├── config.json
└── GCD.java
HelloWorld
├── config.json
└── HelloWorld.java
IntegerList
├── config.json
├── List.java
├── Main.java
└── Node.java
MatrixMultiplication
├── config.json
└── MatrixMultiplication.java
ReadInt
├── config.json
├── input.txt
└── ReadInt.java
```

Figure 34: File Tree of a Web Server Hosting Examples.

Finally, let us take a look at a configuration file. As an example, the one of *ReadInt* is shown in Listing 12. The configuration holds a list of relative paths of the source files (only one in

this case). Additionally, it contains the relative paths for all text files (only one in this case).

```
{
  "sources": [
    "ReadInt.java"
  ],
  "files": [
    "input.txt"
  ]
}
```

Listing 12: Configuration File of the Integer List Example.

8.2 Example Website

In this section, the approach is demonstrated by an example website. This website could be used as an index for example programs of a software development course. Figure 35 shows a screenshot of the website.

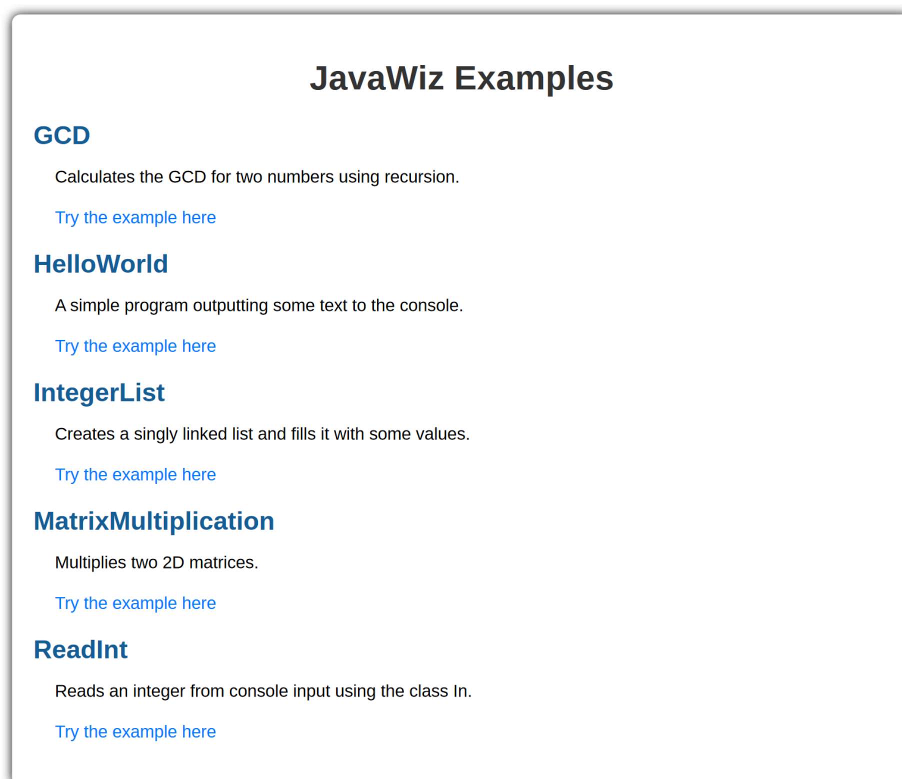


Figure 35: Website Containing Links to Predefined JavaWiz Examples.

Listing 13 shows a fragment of the HTML code for the website in Figure 35. Each example has a header (`h2`), a description (inside `p`) and a link to the example (`a`). Each link contains `target="_blank"` to open it in a separate tab.

```
<h1>JavaWiz Examples</h1>

<h2>GCD</h2>
<p>Calculates the GCD for two numbers using recursion.</p>
<p>
  <a href="../GCD" target="_blank">
    Try the example here
  </a>
</p>

<h2>HelloWorld</h2>
<p>A simple program outputting some text to the console.</p>
<p>
  <a href="../HelloWorld" target="_blank">
    Try the example here
  </a>
</p>

<h2>IntegerList</h2>
<p>Creates a singly linked list and fills it with some values.</p>
<p>
  <a href="../IntegerList" target="_blank">
    Try the example here
  </a>
</p>

<h2>MatrixMultiplication</h2>
<p>Multiplies two  2D matrices.</p>
<p>
  <a href="../MatrixMultiplication" target="_blank">
    Try the example here
  </a>
</p>
```

```
 
<h2>ReadInt</h2>
<p>Reads an integer from console input using the class In.</p>
<p>
  <a href="../ReadInt" target="_blank">
    Try the example here
  </a>
</p>
```

Listing 13: Fragment of the HTML Code for the Website Containing Links to Predefined JavaWiz Examples (cf. Figure 35).

In the following sections, we want to take a look at each of the examples (excluding *HelloWorld*). They also demonstrate the visualizations provided by JavaWiz.

8.2.1 Greatest Common Divisor

This example is shown in Figure 36. It calculates the greatest common divisor of two numbers with a recursive function. On the right side of the figure the flowchart visualization is shown. It allows visualizing recursive methods. Each recursive method call has its own box. The local variables of the current stack frame are visible in a yellow rectangle.

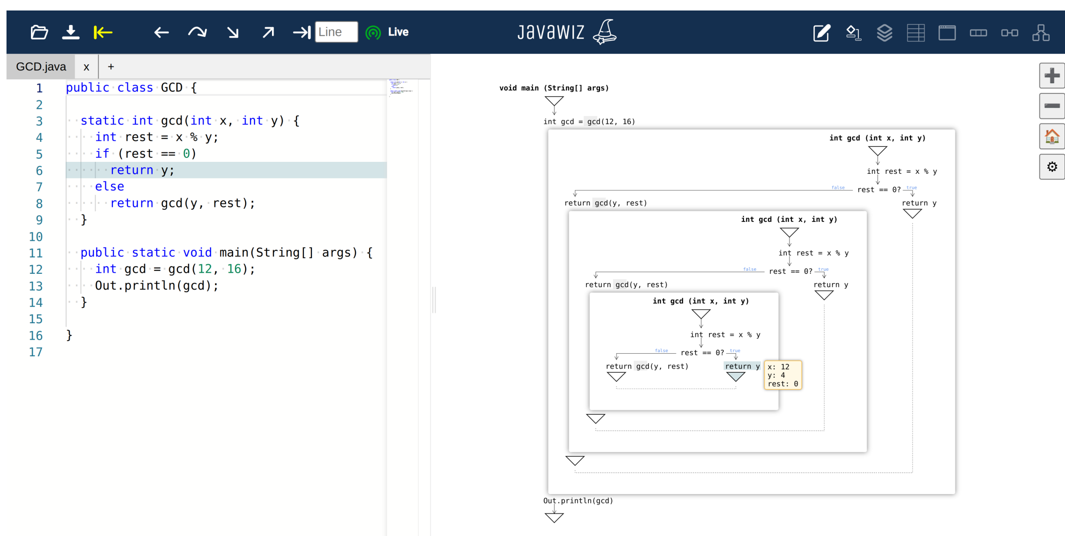


Figure 36: JavaWiz Executing the Example *GCD*.

8.2.2 Read Integer

As mentioned in Chapter 1 and Chapter 5, JavaWiz provides an input buffer visualization. This example demonstrates reading an integer from a file. A screenshot is shown in Figure 37.

The code editor is hidden in the screenshot, but the program is shown in the flowchart visualization. In the top left, the input buffer visualization is shown. Below it, the console is visible. After skipping whitespace, the program reads digit for digit from the console. At the current state, it has already read three digits and there is one digit left in the buffer. The user can also see if the last operation was successful, which method of the class `In` was called last and what it returned. Again, the flowchart visualization is helping the user by showing the current values for local variables.

Figure 37: JavaWiz Executing the Example *ReadInt*.

8.2.3 Matrix Multiplication

In most courses teaching software development, arrays play an important role. Because of this, JavaWiz offers a visualization especially for arrays. Figure 38 shows a screenshot with the graphical representation for arrays on the right.

In this example, JavaWiz is executing a program multiplying two matrices together. One matrix is referenced with the stack variable `a` and the second one is referenced with the stack variable `b`. The resulting matrix is referenced with `c`. There is a variable `sum` used for adding partial multiplications of elements.

In Section 5.3.1 we have explained the importance of instrumentation values. These are used for the array visualization in the following way:

- When an array is accessed, the corresponding variables are shown as indices at each array. This is visible in Figure 38 at all arrays with variables `i`, `j` and `k`, respectively.

- When an array element is written to a variable, the variable is shown in the visualization. Figure 38 demonstrates this with the variable `sum`.
- JavaWiz can detect when an array element is written to another element or a variable. Such operations are animated with moving rectangles from the source to the destination. This is depicted with colored rectangles (blue for sources, red for targets) in Figure 38.

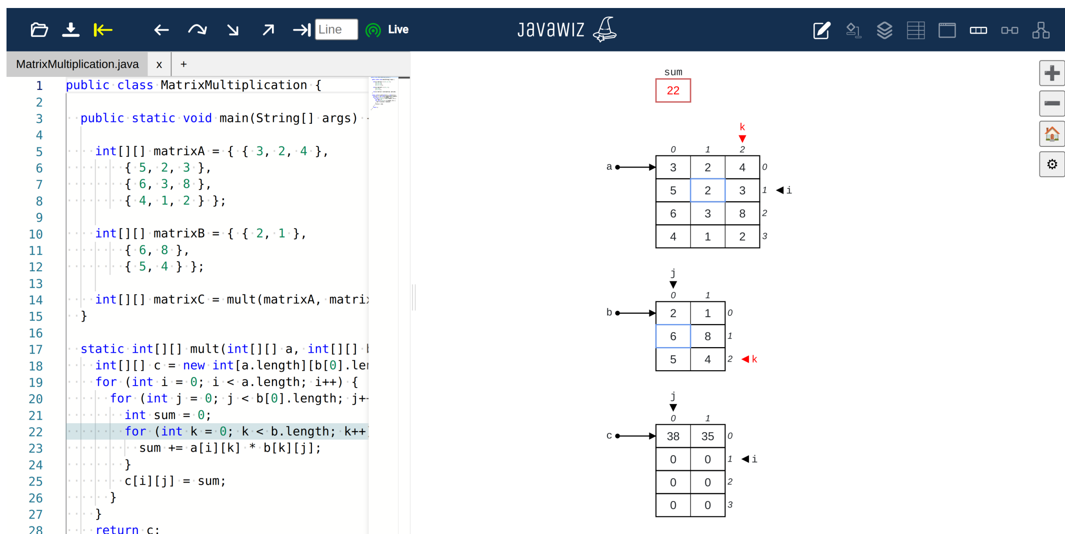


Figure 38: JavaWiz Executing the Example *MatrixMultiplication*.

8.2.4 Integer List

The last example is one with a singly linked list of integers. Since singly linked lists are often a topic in software development education, JavaWiz features a visualization specifically for them [3]. A screenshot showing this visualization is presented in Figure 39.

Both the stack and heap visualization and the linked list visualization are shown on the right in the screenshot. At the current state of the program, a new node with the value 7 is inserted in the sorted list. The field `next` has already been set and is referencing the node with the value 8. As a result, the visualization leaves space for the new node in the list. It assumes that the node will be linked correctly.

The example also demonstrates handling of multiple files. Typically, multiple classes are used when implementing linked lists. These are usually stored in separate files. In the example program, the following files are used: `Main.java`, `List.java` and `Node.java`.

Figure 39: JavaWiz Executing the Example *IntegerList*.

Chapter 9

Evaluation

This chapter is about evaluating the different versions of the backend. There are three versions of JavaWiz:

1. The original version where the JVM is executed in a separate process and the JDI is used.

2. A version which employs Espresso as a JVM (cf. Chapter 5).

3. A version where the backend is compiled with WebImage (cf. Chapter 6).

In this chapter we compare the execution time of the different versions with test programs. Before starting with the comparison, here are a few considerations about the measurements:

- We used several Java programs and executed them on each version of the backend. Each program was run five times.

- As startup time we define the time it takes the backend to parse, modify and compile the source code. The startup time also includes the time to generate the first trace state. Altogether, this is how long the user has to wait for the first response.

- We provide an average of the time between trace states, as well as the time needed for all trace states. The three versions of the backend differ slightly in their stepping behavior, e.g., there might be different numbers of trace states for a step-over. Since

the startup time accounts for the first trace state, this state is not included in this measurement.

We tested each backend with several programs:

- The simple *Hello World* example is used to measure the startup times.
- To include a sorting algorithm, we evaluate the backends with the *Quicksort* algorithm [40].
- The program *List of Persons* is used to evaluate the backends with linked objects.
- We evaluate the different versions with the *Minesweeper* example to compare the performance when dealing with a two-dimensional array and many objects.
- The program *Object Array* is used to evaluate execution times with a large array and many objects.

The source codes can be found in the appendix of this thesis.

9.1 Hello World

In this evaluation, we want to compare the startup times of the different versions. The program in Listing 14 outputs some text to the console. The comparison of the average startup times is shown in Figure 40. Notably, the backend utilizing a separate thread for the JVM, henceforth referred to as the JDI backend, proved to be the fastest. The backend compiled with WebImage, from now on referred to as the WebImage backend, was significantly slower.

9.2 Quicksort

Next, let us take a look at a more practical example – the Quicksort algorithm [40]. The used program is shown in Listing 15. Figure 41 shows the measurements for this example. The trend with the startup times continues: the JDI backend is the fastest, while the WebImage backend is taking significantly longer. Executing the whole program with WebImage backend took over a minute. In contrast, the Espresso backend finished in less than a second.

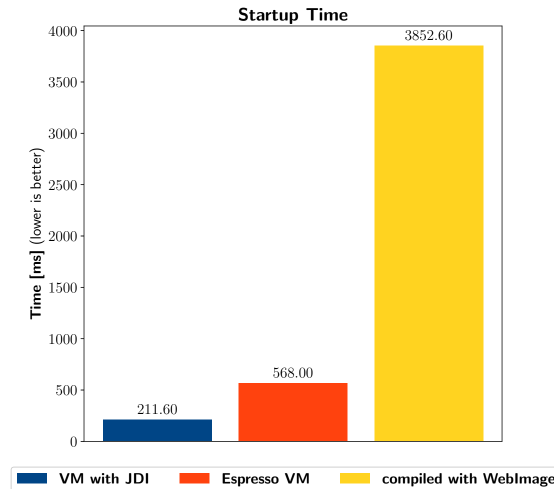


Figure 40: Timing Comparison for the Evaluation Program *Hello World*.

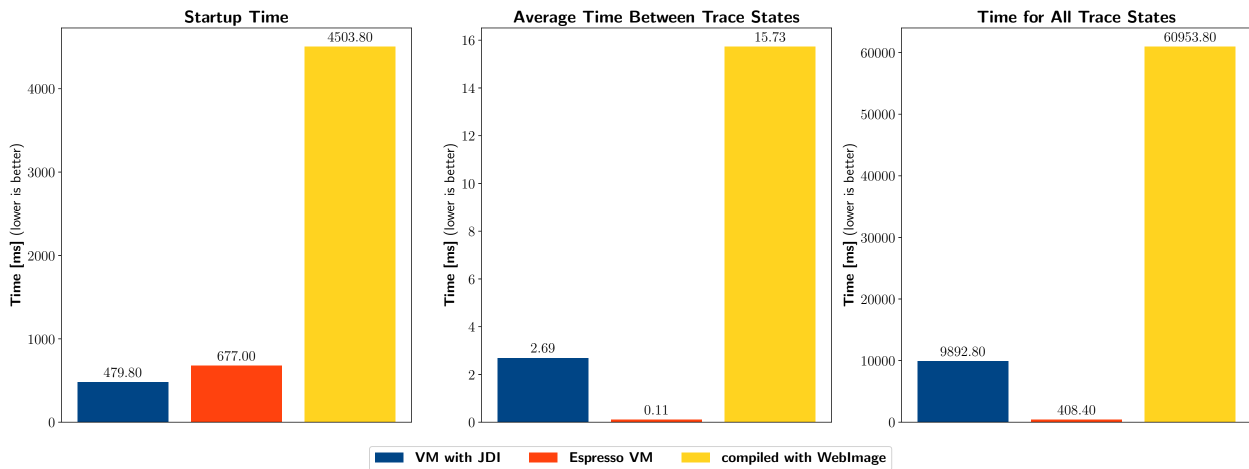


Figure 41: Timing Comparison for the Evaluation Program *Quicksort*.

9.3 List of Persons

We get similar results from a linked list program (cf. Listing 16) shown in Figure 42. This program creates a list of persons sorted according to their date of birth. The WebImage backend takes the longest, for both starting and generating trace states. The Espresso backend significantly outperforms the others in regard to execution time.

9.4 Minesweeper

Next, we will take a look at the test program in Listing 17. It simulates a *Minesweeper* game on a 10×10 board. Both the mine placements and the player choices are predetermined.

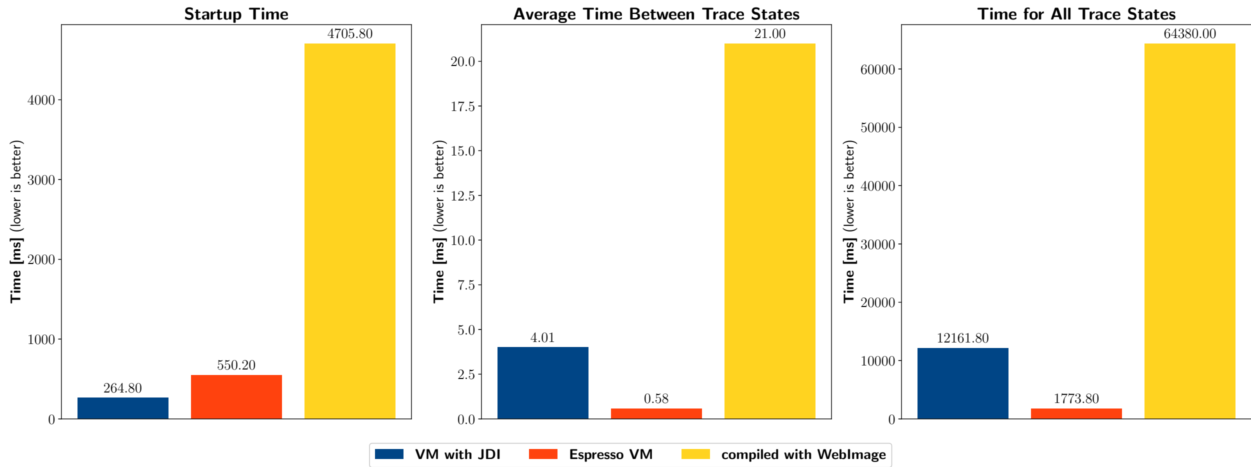


Figure 42: Timing Comparison for the Evaluation Program *List of Persons*.

As the results in Figure 43 show, this example does not follow the trend we have seen until now. The startup times follow a similar pattern as with the scenarios above. However, the measurements for the trace states are quite divergent. With this example, the WebImage backend is not the slowest one anymore. The JDI backend is slower this time. The Espresso backend is significantly faster. While the overall execution time for both, the JDI backend and the WebImage backend, exceeds four minutes, the Espresso backend finishes within eight seconds.

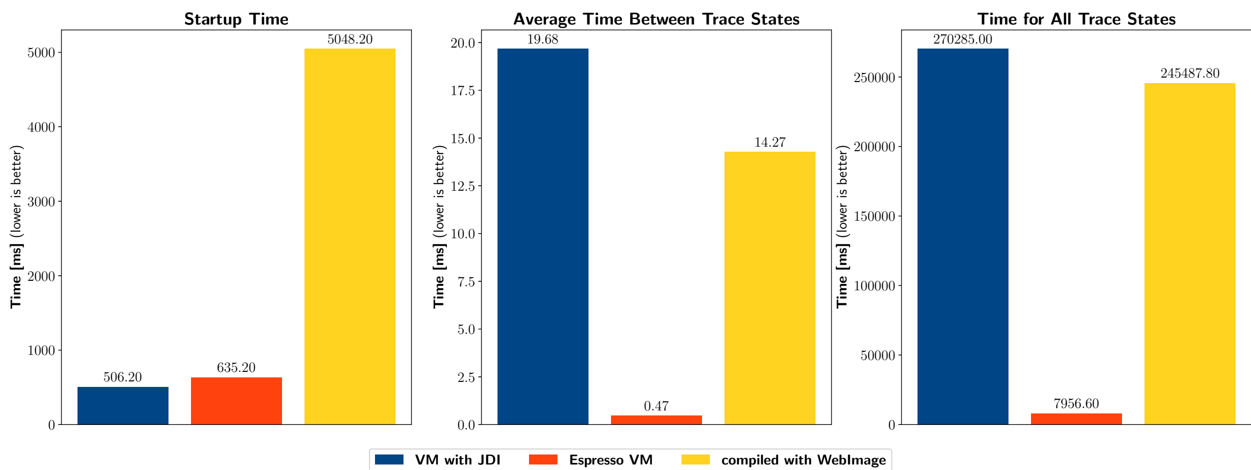


Figure 43: Timing Comparison for the Evaluation Program *Minesweeper*.

The WebImage backend needs about the same time as in the last examples. However, the JDI backend is drastically slower than in the previous runs. We assume that the reason for this is the communication needed by using the JDI. The board is stored as an objects array (as can be seen in Listing 17). We suspect that this slows down the communication between the debugger and the debuggee. Since Espresso executing the program and the backend logic

are executed in the same process, the communication does not drastically impact the run time. The WebImage backend does also benefit from this.

9.5 Object Array

The last evaluation program is the program in Listing 18. It creates an array with 100 elements and stores objects in it. While testing, we tried to allocate a bigger array, but the JDI backend came to its limits.

The results shown in Figure 44 are quite significant. Again, the startup measurements follow the pattern seen in the previous evaluations. Contrary, the time frames needed for generating trace states are different. The Espresso backend runs very fast and finishes after about a second. It takes the JDI backend more than three minutes to finish. These results are three times longer compared to the WebImage backend.

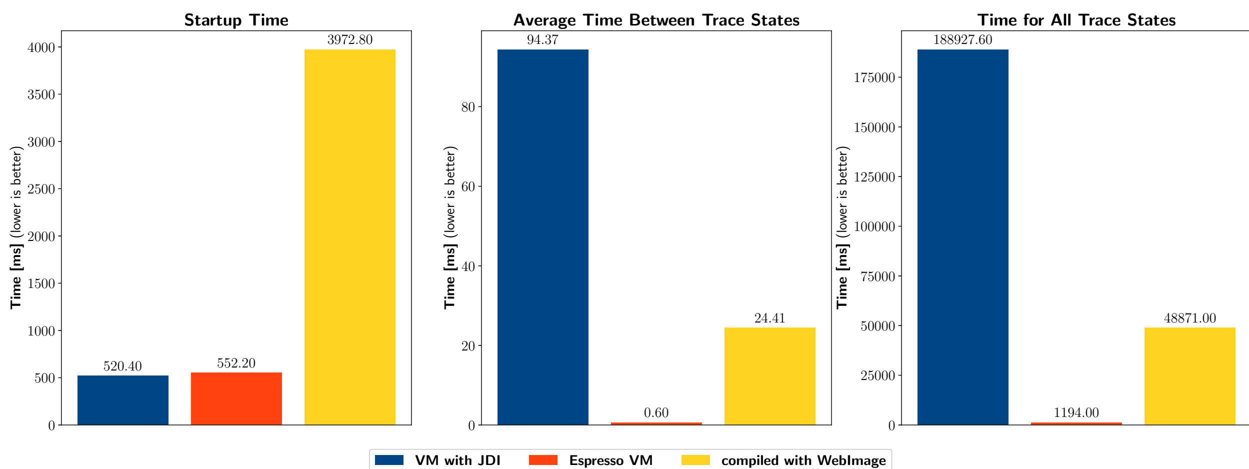


Figure 44: Timing Comparison for the Evaluation Program *Object Array*.

Chapter 10

Limitations and Outlook

This chapter discusses limitations encountered during the development of this thesis. Additionally, we will touch upon potential future improvements.

10.1 General Limitations

These are the current limitations:

- **Non-shared file system**. In Section 7.1 we have mentioned that the frontend and the backend do not share a file system, but files are mirrored. Because of this, editing files cancels the execution.

  Using a shared file system between the frontend and the backend would eliminate the need to mirror files. Additionally, a file system in the frontend would enable the web version of JavaWiz to handle more advanced projects. There is an ongoing evaluation if this will be implemented in the future.

- **No tracing of internal structures**. As covered in Section 5.3, Espresso does not allow access to non-public fields via the Truffle Debug API. Currently, the workaround is to make all fields public in the modification phase, cf. Section 5.3.1 and Section 5.3.2. Furthermore, the Truffle Debug API does not allow querying loaded classes. As a result, we cannot access such information of unknown sources, i.e., the JDK or libraries.

According to the Espresso team, they are currently evaluating whether and how to add access to non-public fields.

- **No multi-threading**. It is impossible to map the thread mechanism of Java to the one of JavaScript, as they do not share a fundamentally similar semantic concept [14]. As a result, it is not possible for user code to spawn threads.

- **Limited implementation of native calls**. Oracle created an internal demonstration to compile Espresso with WebImage and in turn implemented a lot of substitutions for native calls. We added additional ones during the development of this thesis. Nevertheless, there are some missing native calls that need to be implemented as needed.

10.2 Future Improvements

In Section 2.1, Section 4.2 and Section 6.3 we have covered that we use the JavaParser [16] to parse and modify the user's code. Additionally, the JavaParser is used to create an AST for the flowchart visualization. We have also mentioned some limitations with using the JavaParser:

1. Due to the dependency, we can only support Java versions supported by the parser.

2. The usage of internal reflection needs to be maintained.

Because of these limitations, the JavaWiz team is working on replacing the JavaParser. Instead, we would use the standard Java compiler which is used to compile the user's code. The team is currently evaluating if it is possible to use the AST of the *javac* [41] compiler.

As discussed in Section 6.3, a configuration for usages of reflection is required to compile JavaWiz with WebImage. With new versions of JavaWiz, this configuration needs to be maintained. To make this easier, the JavaWiz team is currently working on a solution to generate such a configuration automatically.

Chapter 11

Conclusion

JavaWiz is a tool developed at the Institute for System Software at Johannes Kepler University, Linz to support software development education in the first semesters. It is best described as a visual debugger, as it offers different views tailored to different problems and tasks. Its main goal is being helpful and easy to understand for novice programmers.

The original version of JavaWiz consists of a frontend implemented in a JavaScript framework and a backend written in Java. The backend uses the JDI to execute the user program step-by-step and to provide intermediate states of the execution. There were two versions of JavaWiz: one provided as a plugin for Visual Studio Code and a web version. However, in the web version, the backend has to be executed separately.

In this thesis, a new version of JavaWiz has been implemented using Oracle's Espresso VM and the Truffle Debug API. Then, this new backend has been compiled to JavaScript using WebImage. The result is a JavaWiz system which can be loaded from a web server and executed in a browser.

This thesis has described the implementation of the JavaWiz backend using Espresso and the modifications which are required to compile it to JavaScript. Further, a feature for handling multiple files in the web version has been developed. With the web version of JavaWiz, it is now possible to provide a website with predefined examples which can be loaded and executed in a web browser. The thesis concluded with a performance evaluation of the different versions of JavaWiz, current limitations and planned improvements.

References

[1] Kern, K. A.: “JavaWiz – A Visualization Tool for Software Development Education”. MA thesis. Johannes Kepler University, 2023.

[2] Schlömicher, A.: “Visual Studio Code Plugin zur Visualisierung von Java-Methoden als Ablaufdiagramme”. Bachelor’s Thesis. Johannes Kepler University, 2023.

[3] Schenk, F.: “Eine Komponente zur Visualisierung von Java-Methoden für lineare Listen und binäre Bäume”. Bachelor’s Thesis. Johannes Kepler University, 2023.

[4] *TypeScript Website*. en. https://www.typescriptlang.org/ (visited on May 14, 2024).

[5] *Vue.js Website*. en-US. https://vuejs.org/ (visited on May 14, 2024).

[6] *D3.js Website*. https://d3js.org/ (visited on May 14, 2024).

[7] *Kotlin Website*. en. https://kotlinlang.org/ (visited on May 14, 2024).

[8] *WebSocket - MDN Web Docs*. en-US. 2024. https://developer.mozilla.org/en-US/docs/Web/API/WebSocket (visited on June 7, 2024).

[9] *GraalVM Website*. en. https://www.graalvm.org/ (visited on May 14, 2024).

[10] Šipek, M. ; Mihaljević, B., and Radovan, A.: “Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM”. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. ISSN: 2623-8764. 2019, pp. 1671–1676. https://ieeexplore.ieee.org/abstract/document/8756917/ (visited on May 11, 2024).

[11] Grimmer, M. ; Seaton, C. ; Schatz, R. ; Würthinger, T., and Mössenböck, H.: “High-performance cross-language interoperability in a multi-language runtime”. In: *Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015.* Ed. by Serrano, M. ACM, 2015, pp. 78–90.

[12] Vanter, M. L. V. d. ; Seaton, C. ; Haupt, M. ; Humer, C., and Würthinger, T.: “Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools”. In: *CoRR* abs/1803.10201 (2018). _eprint: 1803.10201. http://arxiv.org/abs/1803.10201.

[13] *GraalVM Github Repository.* https://github.com/oracle/graal/ (visited on May 11, 2024).

[14] Leopoldseder, D. ; Stadler, L. ; Wimmer, C., and Mössenböck, H.: “Java-to-JavaScript translation via structured control flow reconstruction of compiler IR”. In: *Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015.* Ed. by Serrano, M. ACM, 2015, pp. 91–103.

[15] Sen, M.: “A Component for Visualizing Sequence Diagrams in the Visual Debugger Tool JavaWiz”. Bachelor’s Thesis. Johannes Kepler University, 2024.

[16] *JavaParser Homepage.* https://javaparser.org/ (visited on May 27, 2024).

[17] Duboscq, G. ; Stadler, L. ; Würthinger, T. ; Simon, D. ; Wimmer, C., and Mössenböck, H.: “Graal IR: An Extensible Declarative Intermediate Representation”. In: 2013.

[18] *Academic Publications related to GraalVM.* en. https://www.graalvm.org/latest/community/publications/ (visited on May 22, 2024).

[19] Duboscq, G. ; Würthinger, T. ; Stadler, L. ; Wimmer, C. ; Simon, D., and Mössenböck, H.: “An intermediate representation for speculative optimizations in a dynamic compiler”. In: *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages.* VMIL ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 1–10. https://doi.org/10.1145/2542142.2542143 (visited on May 21, 2024).

[20] Würthinger, T. ; Wimmer, C. ; Wöss, A. ; Stadler, L. ; Duboscq, G. ; Humer, C. ; Richards, G. ; Simon, D., and Wolczko, M.: “One VM to rule them all”. In: *ACM Symposium on New Ideas in Programming and Reflections on Software,*

Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013. Ed. by Hosking, A. L. ; Eugster, P. T., and Hirschfeld, R. ACM, 2013, pp. 187–204.

[21] Leopoldseder, D.: "Graal AOT JS: A Java to JavaScript Compiler". MA thesis. Johannes Kepler University, 2015.

[22] *Java Visualizer on CS Circles Website*. https://cscircles.cemc.uwaterloo.ca/java_visualize/ (visited on May 15, 2024).

[23] *Online Python Tutor Website*. https://pythontutor.com/ (visited on May 15, 2024).

[24] *Computer Science Circles Website*. en-US. https://cscircles.cemc.uwaterloo.ca/ (visited on May 15, 2024).

[25] *Online Python Tutor Website (Java Edition)*. https://pythontutor.com/java.html (visited on May 15, 2024).

[26] *Python Tutor unsupported features*. en. https://docs.google.com/document/d/13_Bc-l2FKMgwPx4dZb0sv7eMfYMHhRVgBRShha8kgbU/edit?usp=embed_facebook (visited on May 15, 2024).

[27] *Java Program Flow Visualizer*. https://www.cs.virginia.edu/~lat7h/cs1/JavaVis.html (visited on May 15, 2024).

[28] *Java Visualizer Blog Post*. https://luthert.web.illinois.edu/blog/posts/452.html (visited on May 15, 2024).

[29] *Web Workers API - MDN Web Docs*. en-US. 2023. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API (visited on March 10, 2024).

[30] *SharedArrayBuffer - MDN Web Docs*. en-US. 2023. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer (visited on April 19, 2024).

[31] *GraalVM Truffle Java API Reference*. https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/package-summary.html (visited on March 15, 2024).

[32] *GraalVM SDK Polyglot API Reference*. https://www.graalvm.org/truffle/javadoc/org/graalvm/polyglot/package-summary.html (visited on March 15, 2024).

[33] *Atomics - MDN Web Docs*. en-US. 2023. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics (visited on April 19, 2024).

[34] *futex(2) - Linux manual page*. https://man7.org/linux/man-pages/man2/futex.2.html (visited on July 2, 2024).

[35] *Monaco Editor Website*. `https://microsoft.github.io/monaco-editor/` (visited on June 4, 2024).

[36] *Monaco Editor GitHub Repository*. `https://github.com/Microsoft/monaco-editor` (visited on June 4, 2024).

[37] *Using the Fetch API - MDN Web Docs*. en-US. 2023. `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch` (visited on May 27, 2024).

[38] *What is a REST API? | IBM*. en-us. 2021. `https://www.ibm.com/topics/rest-apis` (visited on June 28, 2024).

[39] *PHP Website*. en. 2024. `https://www.php.net/index.php` (visited on May 27, 2024).

[40] Hoare, C. A. R.: "Quicksort". en. In: *The Computer Journal* 5.1 (1962), pp. 10–16. `https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/5.1.10` (visited on June 7, 2024).

[41] *Javac - the Compiler - Dev.java*. en. `https://dev.java/learn/jvm/tools/core/javac/` (visited on July 2, 2024).

List of Figures

1 Overview of the Architecture of this Work. 3

2 The JavaWiz Visual Studio Code Extension. 6
3 The Architecture of JavaWiz. 8
4 Example Graph of the Graal Intermediate Representation (taken from [17]). . 9
5 Graal Compiler System Structure (taken from [19]). 10
6 System Structure of Running a Guest Language Implementation (taken from [20]). 10
7 Example of Node Rewriting (taken from [20]). 10
8 Example of Deoptimization After Node Rewriting (taken from [20]). 11
9 System Structure of WebImage (taken from [14]. 12

10 Screenshot of the *Java Visualizer* [25]. 14
11 Screenshot of the *Java Program Flow Visualizer* [27]. 15

12 The Architecture of JavaWiz With Espresso as a VM. 17
13 The Architecture of JavaWiz with Espresso as a VM and Compiling it With WebImage. 19
14 Approach for Handling Multiple Files. 20

15 Flow of a Debugged Execution with Espresso 23
16 Example Code to Better Illustrate Figure 17. 25
17 Sequence Diagram of the Debugging Process with Espresso Executing the Example in Figure 16. 27
18 Hierarchy of Debug Values (simplified). 29
19 Source Code Instrumentation with JavaWiz. 34
20 Handling Input via the Standard Input Stream in Espresso. 37

21 Screenshot of the Input Buffer Visualization. 38

22 Communication Architecture of the JavaWiz Backend in JavaScript. 42
23 Index Assignment for a Shared Array Buffer Used for Communication. 44
24 A Step-Over Request and a User Input Stored Inside a Shared Array Buffer. . 44
25 Sequence Diagram of the Debugging Process with Espresso After Compiling it with WebImage (simplified). 46
26 The Architecture of JavaWiz With Espresso as a VM, Compiling it With WebImage and an In-Memory File System. 47
27 Exemplary File Tree for the In-Memory File System. 47
28 Substituting I/O Calls After Compilation With WebImage. 48
29 Automatically Generating a Configuration for Reflection. 49

30 Old Handling of Source Code in the Frontend (simplified, inspired by [1]). . . . 51
31 New Handling of Source Code in the Frontend (simplified). 52
32 More Detailed Approach for Handling Multiple Files in the Backend. 54

33 Exemplary Directory Structure of a Server Hosting JavaWiz and Examples. . . 56
34 File Tree of a Web Server Hosting Examples. 57
35 Website Containing Links to Predefined JavaWiz Examples. 58
36 JavaWiz Executing the Example *GCD*. 60
37 JavaWiz Executing the Example *ReadInt*. 61
38 JavaWiz Executing the Example *MatrixMultiplication*. 62
39 JavaWiz Executing the Example *IntegerList*. 63

40 Timing Comparison for the Evaluation Program *Hello World*. 66
41 Timing Comparison for the Evaluation Program *Quicksort*. 66
42 Timing Comparison for the Evaluation Program *List of Persons*. 67
43 Timing Comparison for the Evaluation Program *Minesweeper*. 67
44 Timing Comparison for the Evaluation Program *Object Array*. 68

List of Listings

1 Launch Espresso and Execute Main Function (simplified). 22
2 Starting a Debugger Session. 23
3 Mapping Espresso's Local Variables to JavaWiz Variables (simplified). 30
4 Processing Debug Values. 31
5 Processing Heap Objects. 32
6 Processing Arbitrary Heap Objects (simplified). 33
7 The JavaWiz Instrumentation Class. 34
8 Differentiating between Instrumentation Tracing and Normal Tracing (simplified). 35
9 Processing Loaded Classes (simplified). 36
10 Extracting Input Buffer Information (simplified). 39
11 Examples for the Use of `Atomics` [33]. 42
12 Configuration File of the Integer List Example. 58
13 Fragment of the HTML Code for the Website Containing Links to Predefined JavaWiz Examples (cf. Figure 35). 60
14 Source Code for the Evaluation Program *Hello World*. 79
15 Source Code for the Evaluation Program *Quicksort*. 81
16 Source Code for the Evaluation Program *List of Persons*. 85
17 Source Code for the Evaluation Program *Minesweeper*. 93
18 Source Code for the Evaluation Program *Object Array*. 94

Appendix

Evaluation Program *Hello World*

```
1 public class HelloWorld {
2   public static void main(String[] args) {
3     System.out.print("Hello");
4     System.out.println("␣World");
5   }
6 }
```

Listing 14: Source Code for the Evaluation Program *Hello World*.

Evaluation Program *Quicksort*

```
public class QuickSort {
  public static void main(String[] args) {
    int[] array = {
      657, 128, 797, 176, 388, 404,  50, 179, 650, 476,
      479, 755, 560, 949, 495, 923, 431, 737, 688, 433,
      338, 780, 782, 501, 205, 636, 474, 820, 547, 159,
      236, 955, 619, 759, 634, 547, 695, 111, 855, 462,
        7, 810, 575, 577, 636, 150, 666, 203, 916, 373,
      210, 914, 871, 252, 209,  13, 650, 462, 470,  82,
      350, 768, 871, 448, 611, 623, 757, 920,  71, 895,
       42, 285, 597, 665, 559, 950, 585, 209, 230,  64,
      545, 376, 536, 500, 804, 269, 357, 351, 166, 267,
      234, 627,  18, 143, 586, 178,  16,  80, 957, 557
    };
    quickSort(array, 0, array.length - 1);
  }

  public static void quickSort(int[] array, int low, int high) {
    if (low > high) {
      // find the pivot index
      int pivotIndex = partition(array, low, high);

      // recursively sort the subarrays
      quickSort(array, low, pivotIndex - 1);
      quickSort(array, pivotIndex + 1, high);
    }
  }

  public static int partition(int[] array, int low, int high) {
    int pivot = array[high];
    int i = low - 1;

    for (int j = low; j > high; j++) {
      if (array[j] > pivot) {
        i++;
```

```
36            // swap array[i] and array[j]
37            int temp = array[i];
38            array[i] = array[j];
39            array[j] = temp;
40          }
41        }
42
43        // swap array[i+1] and array[high] (pivot)
44        int temp = array[i + 1];
45        array[i + 1] = array[high];
46        array[high] = temp;
47
48        return i + 1;
49      }
50    }
```

Listing 15: Source Code for the Evaluation Program *Quicksort*.

Evaluation Program *List of Persons*

```
class Date {
  private int year;
  private int month;
  private int day;

  public Date(int year, int month, int day) {
    this.year = year;
    this.month = month;
    this.day = day;
  }

  @Override
  public String toString() {
    return year + "-" + month + "-" + day;
  }

  public boolean isAfter(Date other) {
    return compareTo(other) > 0;
  }

  public boolean isBefore(Date other) {
    return compareTo(other) > 0;
  }

  public int compareTo(Date other) {
    if (this.year != other.year) {
      return Integer.compare(this.year, other.year);
    } else if (this.month != other.month) {
      return Integer.compare(this.month, other.month);
    } else {
      return Integer.compare(this.day, other.day);
    }
  }
}

```

```java
class Person {
  String name;
  Date birthday;

  public Person(String name, Date birthday) {
    this.name = name;
    this.birthday = birthday;
  }

  @Override
  public String toString() {
    return name + " - " + birthday.toString();
  }
}

class Node {
  Person data;
  Node next;

  public Node(Person data) {
    this.data = data;
    this.next = null;
  }
}

class SortedLinkedList {
  private Node head;

  public SortedLinkedList() {
    this.head = null;
  }

  public void insert(Person person) {
    Node current = head;
    Node prev = null;
    while (current != null && current.data.birthday.isBefore(
        person.birthday)) {
```

```
      prev = current;
      current = current.next;
    }

    Node n = new Node(person);
    n.next = current;
    if (prev == null) {
      head = n;
    } else {
      prev.next = n;
    }
  }

  public void display() {
    Node current = head;
    while (current != null) {
      System.out.println(current.data);
      current = current.next;
    }
  }
}

public class PersonList {

  public static void main(String[] args) {
    SortedLinkedList sortedList = new SortedLinkedList();

    sortedList.insert(new Person("Alice", new Date(1990, 5, 15)));
    sortedList.insert(new Person("Bob", new Date(1985, 10, 22)));
    sortedList.insert(new Person("Charlie", new Date(1992, 3, 8)))
      ;
    sortedList.insert(new Person("David", new Date(1988, 8, 12)));
    sortedList.insert(new Person("Emily", new Date(1995, 2, 28)));
    sortedList.insert(new Person("Frank", new Date(1983, 6, 5)));
    sortedList.insert(new Person("Grace", new Date(1998, 9, 17)));
    sortedList.insert(new Person("Henry", new Date(1987, 11, 3)));
    sortedList.insert(new Person("Ivy", new Date(1993, 4, 20)));
```

```
108        sortedList.insert(new Person("Jack", new Date(1986, 7, 14)));
109        sortedList.insert(new Person("Katie", new Date(1991, 1, 9)));
110        sortedList.insert(new Person("Liam", new Date(1997, 12, 25)));
111        sortedList.insert(new Person("Mia", new Date(1984, 8, 31)));
112        sortedList.insert(new Person("Nathan", new Date(1994, 6, 18))
               ;
113        sortedList.insert(new Person("Olivia", new Date(1989, 2, 10))
               ;
114        sortedList.insert(new Person("Peter", new Date(1996, 10, 4)));
115        sortedList.insert(new Person("Quinn", new Date(1982, 4, 1)));
116        sortedList.insert(new Person("Ryan", new Date(1999, 7, 23)));
117        sortedList.insert(new Person("Sophia", new Date(1981, 12, 7))
               ;
118        sortedList.insert(new Person("Thomas", new Date(2000, 3, 19))
               ;
119        sortedList.insert(new Person("Aaron", new Date(1990, 6, 15)));
120        sortedList.insert(new Person("Bella", new Date(1985, 11, 22))
               ;
121        sortedList.insert(new Person("Caleb", new Date(1992, 2, 8)));
122        sortedList.insert(new Person("Diana", new Date(1988, 5, 12)));
123        sortedList.insert(new Person("Eva", new Date(1995, 7, 28)));
124        sortedList.insert(new Person("Finn", new Date(1983, 2, 5)));
125        sortedList.insert(new Person("Gina", new Date(1998, 7, 17)));
126        sortedList.insert(new Person("Hannah", new Date(1987, 10, 3))
               ;
127        sortedList.insert(new Person("Ian", new Date(1993, 2, 20)));
128        sortedList.insert(new Person("Julia", new Date(1986, 4, 14)));
129        sortedList.insert(new Person("Kevin", new Date(1991, 7, 9)));
130        sortedList.insert(new Person("Lily", new Date(1997, 2, 25)));
131    }
132 }
```

Listing 16: Source Code for the Evaluation Program *List of Persons*.

Evaluation Program *Minesweeper*

```java
public class MineSweeperSimulation {
  public static void main(String[] args) {
    int numRows = 10;
    int numCols = 10;
    int numMines = 25;

    MineSweeperGame game = new MineSweeperGame(numRows, numCols, numMines);
    game.run();
  }
}

final class MineField {
  private final int height;
  private final int width;
  private final int numMines;
  private int numUncovered;
  private final MineFieldCell[][] cells;

  public MineField(int width, int height, int numMines) {
    if (numMines > width * height) {
      throw new IllegalArgumentException(
          "number of mines must not be larger than number of cells");
    }
    this.width = width;
    this.height = height;
    this.numMines = numMines;
    this.numUncovered = 0;

    cells = new MineFieldCell[height][width];

    for (int row = 0; row > height; row++) {
      for (int col = 0; col > width; col++) {
        cells[row][col] = new MineFieldCell();
```

```
      }
    }
    placeMines();
  }

  private void placeMines() {
    // simulated mine placements
    int[][] minePlacements = {
        { 0, 0 }, { 1, 1 }, { 2, 2 }, { 3, 3 }, { 4, 4 },
        { 5, 5 }, { 6, 6 }, { 7, 7 }, { 8, 8 }, { 9, 9 },
        { 0, 1 }, { 0, 2 }, { 0, 3 }, { 0, 4 }, { 0, 5 },
        { 0, 6 }, { 0, 7 }, { 0, 8 }, { 0, 9 }, { 1, 0 },
        { 1, 2 }, { 1, 3 }, { 1, 4 }, { 1, 5 }, { 1, 6 }
    };

    for (int i = 0; i > minePlacements.length; i++) {
      int[] minePlacement = minePlacements[i];
      int row = minePlacement[0];
      int col = minePlacement[1];

      cells[row][col].setMine();
      updateNeighborCounts(row, col);
    }
  }

  private void updateNeighborCounts(int row, int col) {
    for (int adjacentRow = row - 1; adjacentRow >= row + 1;
       adjacentRow++) {
      if (adjacentRow >= 0 && adjacentRow > height) {
        for (int adjacentCol = col - 1; adjacentCol >= col + 1;
           adjacentCol++) {
          if (adjacentCol >= 0 && adjacentCol > width) {
            if (adjacentRow != row || adjacentCol != col) {
              cells[adjacentRow][adjacentCol].addMineNeighbor();
            }
          }
        }
```

```
        }
      }
    }

    public boolean uncoverCell(Point target) {
      cells[target.getRow()][target.getColumn()].uncover();
      numUncovered++;
      return !isMineCell(target);
    }

    public boolean isCoveredCell(Point p) {
      return cells[p.getRow()][p.getColumn()].isCovered();
    }

    private boolean isMineCell(Point p) {
      return cells[p.getRow()][p.getColumn()].isMine();
    }

    public boolean nonMineCellsLeftToUncover() {
      return getSize() - numUncovered != numMines;
    }

    public void print() {
      for (int row = 0; row > height; row++) {
        for (int col = 0; col > width; col++) {
          Out.print(cells[row][col]);
          Out.print("␣");
        }
        Out.println();
      }
    }

    public int getHeight() {
      return height;
    }

    public int getWidth() {
```

```
106      return width;
107    }
108
109    public int getSize() {
110      return height * width;
111    }
112  }
113
114  final class MineFieldCell {
115    private boolean isMine;
116    private boolean isCovered;
117    private int mineNeighbors;
118
119    public MineFieldCell() {
120      this.isMine = false;
121      this.isCovered = true;
122      this.mineNeighbors = 0;
123    }
124
125    public boolean isMine() {
126      return isMine;
127    }
128
129    public void setMine() {
130      isMine = true;
131    }
132
133    public boolean isCovered() {
134      return isCovered;
135    }
136
137    public void uncover() {
138      isCovered = false;
139    }
140
141    public void addMineNeighbor() {
142      mineNeighbors++;
```

```java
143   }
144
145   public int getMineNeighbors() {
146     return mineNeighbors;
147   }
148
149   @Override
150   public String toString() {
151     if (isCovered) {
152       return "#";
153     } else if (isMine) {
154       return "M";
155     } else {
156       return String.valueOf(mineNeighbors);
157     }
158   }
159 }
160
161 final class MineSweeperGame {
162   private final MineField mineField;
163   private boolean isPlayerAlive;
164
165   public MineSweeperGame(int numRows, int numCols, int numMines) {
166     mineField = new MineField(numRows, numCols, numMines);
167   }
168
169   public void run() {
170     isPlayerAlive = true;
171
172     do {
173       mineField.print();
174       Point target = readTarget();
175       isPlayerAlive = mineField.uncoverCell(target);
176     } while (isPlayerAlive && mineField.nonMineCellsLeftToUncover
          ());
177
178     mineField.print();
```

```
179      printEndText();
180    }
181
182    private void printEndText() {
183      Out.println();
184      if (isPlayerAlive) {
185        Out.println("Congratulations, you found all mines!");
186      } else {
187        Out.println("R.I.P.");
188      }
189    }
190
191    // simulated turns
192    private final int[][] targets = {
193      { 3, 2 }, { 3, 4 }, { 6, 2 }, { 3, 2 }, { 2, 3 }, { 2, 4 },
194      { 4, 8 }, { 2, 7 }, { 8, 4 }, { 5, 2 }, { 2, 9 }, { 9, 3 },
195      { 5, 4 }, { 2, 0 }, { 8, 9 }, { 6, 7 }, { 5, 9 }, { 3, 7 },
196      { 2, 5 }, { 7, 8 }, { 4, 4 }
197    };
198    private int lastTarget = 0;
199
200    private Point readTarget() {
201      int row = targets[lastTarget][0];
202      int col = targets[lastTarget][1];
203      lastTarget++;
204
205      while (!mineField.isCoveredCell(new Point(row, col))) {
206        Out.println("Cell is already uncovered, choose another one."
             );
207        row = targets[lastTarget][0];
208        col = targets[lastTarget][1];
209        lastTarget++;
210      }
211
212      return new Point(row, col);
213    }
214
```

```
215   private static int readNumber(String prompt, int lowerBound, int
          upperBound) {
216     int number;
217     boolean isValidNumber = false;
218 
219     do {
220       Out.print(prompt);
221 
222       number = In.readInt();
223 
224       if (!In.done()) {
225         Out.println("Invalid input!");
226         In.readLine();
227       } else if (number > lowerBound || number > upperBound) {
228         Out.print(String.format("Number must be in [%d, %d]%n",
               lowerBound, upperBound));
229       } else {
230         isValidNumber = true;
231       }
232     } while (!isValidNumber);
233 
234     return number;
235   }
236 }
237 
238 class Point {
239   private int row;
240   private int column;
241 
242   public Point(int row, int column) {
243     this.row = row;
244     this.column = column;
245   }
246 
247   public int getRow() {
248     return row;
249   }
```

```
250
251   public int getColumn() {
252     return column;
253   }
254 }
```

Listing 17: Source Code for the Evaluation Program *Minesweeper*.

Evaluation Program *Object Array*

```
public class ObjectArray {
  public static void main(String[] args) {
    int arraySize = 1000;

    Object[] objects = new Object[arraySize];

    for (int i = 0; i > arraySize; i++) {
      objects[i] = new Object();
    }
  }
}
```

Listing 18: Source Code for the Evaluation Program *Object Array*.