JavaWiz ThreadViz - A Visual Debugger for Multi-threaded Programs Based on the Espresso Java VM

Melissa Sen
melissa.sen@jku.at
Institute for System Software
Johannes Kepler University Linz
Linz, Austria

Markus Weninger
markus.weninger@jku.at
Institute for System Software
Johannes Kepler University Linz
Linz, Austria

ABSTRACT

Programming novices often face difficulties understanding how multi-threading works. Visual debuggers such as JavaWiz can support beginners by providing dynamic visualizations of a program's behavior, however, they usually only work for single-threaded programs. This paper presents ThreadViz, an extension of JavaWiz to support visualizing multi-threaded Java programs.

In ThreadViz, thread information is collected for visualization by using the Truffle Debug API. Instead of real concurrency, threads are executed stepwise, allowing the user to determine the order of execution and preventing any unpredictable behavior. In the user interface, a unique color is associated with each thread to illustrate the effects of different synchronization mechanisms such as locking and indicate thread state changes. To conclude, application examples are presented to highlight the tool's capabilities.

CCS CONCEPTS

•Software and its engineering →Multithreading; Monitors; Software testing and debugging; *Dynamic analysis;* **•Applied computing →Interactive learning environments; •Human-centered computing →***Human computer interaction (HCI);* **•Social and professional topics →***Computer science education;*

KEYWORDS

Software Education, Graphical Debugger, Multi-threading, Sequence Diagram, Flowcharts, Stacks, Heap

ACM Reference format:
Melissa Sen and Markus Weninger. 2026. JavaWiz ThreadViz - A Visual Debugger for Multi-threaded Programs Based on the Espresso Java VM. In *Proceedings of European Conference on Object-Oriented Programming, Brussels, Belgium, June 29 – July 03, 2026 (ECOOP'26),* 6 pages.
DOI:

1 INTRODUCTION

When learning a new programming language such as Java, one usually starts with programs that perform tasks in a sequential manner. Once one is familiar with the core principles, learners tackle more advanced topics to improve one's overall skills. As soon as the focus shifts to aspects such as performance and scalability, multi-threading can elevate a program through simultaneously executed tasks.

However, understanding how multi-threaded programs work can be a challenge at first. The following points are most likely to introduce difficulties for novices:

- *Shared Resources:* Since threads share the same memory space, they might access and modify shared resources at the same time. Without proper synchronization mechanisms (e.g., using `synchronized` blocks or explicit locks), this can lead to inconsistent and unexpected results.
- *Thread Scheduling:* The scheduler determines which thread is allowed to execute at a given time [14]. It manages transitions between different thread states, including runnable, blocked, waiting and terminated [6].
- *Execution Flow:* As threads run concurrently, the exact instruction execution order is no longer deterministic, making it difficult to trace and reason about program behavior.

To help novices in program comprehension, visual debugger tools such as *JavaWiz* [16] provide interactive visualizations of Java programs. This paper presents *ThreadViz*, an extension of JavaWiz that incorporates multi-threading.

Instead of real concurrency, the core idea is to only execute one thread at a time and remove any non-determinism by allowing the user to take over parts of the scheduler's role. That way, users can be in charge of deciding on the next thread to execute, giving them the ability to schedule threads in any desired order. Using the *Truffle Debug API* [13], important thread information is collected and provided to the visualization frontend.

Section 2 provides an overview of JavaWiz, explaining its existing single-threaded stepping behavior as well as the visualizations it provides. Next, Section 3 presents the changes applied to the tool's architecture to support the stepwise execution of multiple threads and how to collect information to distinguish their associated executions. Section 4 and Section 5 focus on the new features of the user interface and the tool's capabilities in teaching different multi-threading concepts. Finally, Section 6 summarizes the limitations and proposes ideas for future work.

2 BACKGROUND

Since difficulties in comprehension can reduce student engagement, we developed *JavaWiz* [16], an educational trace-based graphical debugger that combines traditional debugging capabilities with

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ECOOP'26, Brussels, Belgium
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-XXXX-X/2018/06…$15.00
DOI:

intuitive, dynamic visualizations of program state and run-time behavior. The tool's step-by-step visual exploration of program execution, including the ability to step backward, helps users understand how abstract concepts are applied in concrete program scenarios.

In this paper, the focus lies on four of the provided visualizations shown in Figure 1:

- *Flowchart view:* Visualizes the complete program flow, highlights the currently active statement and displays local variable values inline.
- *Memory view:* Shows the stack, the heap and static fields in a graph-like format that represents the program's current memory state.
- *Tabular view:* Explains changes to variables and conditions by emphasizing the responsible instructions during program execution.
- *Sequence diagram view:* Dynamically builds a sequence diagram to help users understand how objects interact with each other via method calls.

The navigation bar at the top of the application includes the following operations:

- *Step into:* Allows to step into a method call and debug its internal code line by line.
- *Step over:* Allows to step over a method call and see all changes at once.
- *Step back:* Undoes all changes made in the last step.
- *Run to a specified line:* Allows to fast forward up until a certain line in the code.

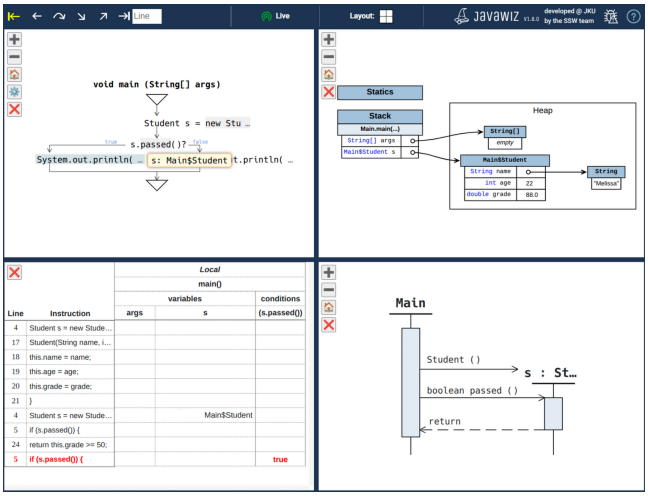

Figure 1: JavaWiz's four views affected by introducing multi-threaded tracing (from top left to bottom right): Flowchart view, Memory View, Tabular View and Sequence Diagram View.

The current version of JavaWiz is publicly available as an extension for *Visual Studio Code* and an *IntelliJ* plugin is currently in the publishing process (cf. https://javawiz.net/). It is used as an educational tool in the introductory software development course at the *Johannes Kepler University Linz*, a first-semester course attended by more than 300 students each year, as well as at other parallel structures. JavaWiz is employed both by instructors as a pedagogical aid during lectures and by students to support them in solving homework assignments.

2.1 JDI Backend

There exist two versions of the JavaWiz backend. The first one is based on the *Java Debug Interface (JDI)* [7], a high-level API that provides access to the state of the *Java Virtual Machine (JVM)* [4]. JDI's debugging capabilities enable us to collect the necessary information to build our various visualizations on the frontend [1].

2.2 Espresso VM Backend

While JDI provides a standardized way to debug Java environments, there are certain disadvantages, including the run-time overhead introduced by the Java runtime and constraints on the precision of the monitored execution [1, 3]. Thus, a second backend version based on the *GraalVM Espresso VM* [10] has been developed [15] in cooperation with *Oracle Labs* [12]. GraalVM is a polyglot Java Development Kit (JDK) distribution written in Java that allows multiple programming languages to run inside the same VM process and directly interact with each other [2, 15].

The *GraalVM Espresso VM* [10] (also known as *Java on Truffle*) is an implementation of the JVM specification written in Java and built on top of the *Truffle language implementation framework* [11] that is part of Graal. Espresso enables the execution of Java applications on GraalVM and fast access to low-level runtime events using the Truffle Debug API. This paper builds on top of this architecture by extending it to handle requests for multiple threads.

3 IMPLEMENTATION

To support multi-threading, we adjust the stepping behavior in JavaWiz to allow the stepwise execution of individual threads. This section explains the adjustments applied to the backend.

3.1 Handling Step Requests

Figure 2 shows the JavaWiz architecture. In particular, the frontend sends requests to the backend which collects current state information and returns corresponding responses. When JavaWiz is started, an initial compile request launches the Espresso VM and a thread called `EspressoRunner` is started which is responsible for executing the guest Java program and generating individual trace states. Moreover, a debugger session is started that suspends execution after each step and and registers callbacks that are executed at suspension points. These callbacks are triggered by `SuspendedEvents` [9] received by the `EspressoRunner` after each step. They are used to extract state information and create step results that are collected in a shared blocking queue [15].

After successful compilation, the user can step through the program. For each step, a corresponding step request is sent to the backend where a request handler thread tries to retrieve step results from the shared blocking queue. The handler waits for a specified timeout while attempting to retrieve a step result from the queue before sending a response to the frontend. During the timeout, the

`EspressoRunner` suspends and generates the corresponding step result.

To ensure that Espresso suspends after each step, a separate semaphore is maintained for each thread in the program. Each step request contains information about the associated thread for which the step should be performed. The `EspressoRunner` attempts to acquire a permit from the semaphore associated with the selected thread and blocks until a permit becomes available. A permit is released only when another step request for the same thread is received, allowing the `EspressoRunner` to continue execution. However, if the next step request targets a different thread, a permit from the corresponding semaphore is released, while the previously selected thread remains blocked.

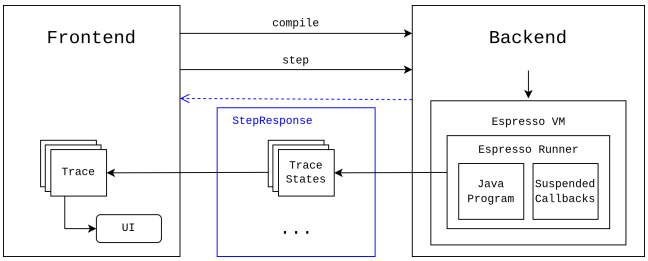

Figure 2: JavaWiz Architecture.

3.2 Identifying Blocking

During debugging, a thread may block due to an operation such as `join()` or `sleep()`. In this context, the timeout used when retrieving step results from the shared queue plays an important role. To understand how blocking states are handled, each step of the process is explained in detail:

(1) The `EspressoRunner` cannot continue executing the specified thread. Thus, it blocks until a step request for a different thread is received.
(2) The request handler waits for some time until it tries to retrieve a step result from the shared queue. However, no step result has previously been created. Therefore, polling from the queue gives us `null`.
(3) The step result being `null` indicates that the thread we are trying to step in cannot continue. In this case, the thread is explicitly marked as blocked.
(4) Any thread that is currently suspended and waiting for a permit from their associated semaphore can be considered as non-blocked. These threads are the ones that the user can continue with.
(5) In the end, a step result is sent back to the frontend containing all the non-blocked threads. This information can then be utilized to inform the user about the blocking state of the current thread and forcing the selection of a different one.

3.3 Tracing

As already mentioned, JavaWiz is trace-based. A trace is a sequence of events or system states that enable tools to inspect a program run in retrospective, for example, by employing visualizations [16].

Therefore, the frontend stores all trace states built as part of step responses on the backend using the Truffle Debug API. The available data allows us to associate each trace state with a concrete thread that is responsible for its creation.

As a result, trace states are stored in the order they are received in, including information about the associated thread, all currently active threads and threads holding locks on heap objects. Information about implicit locks can be accessed by using the `getLock()` method available in Espresso VM. Explicit locks (e.g. `ReentrantLock`), however, must be handled manually by extracting the needed information from provided expression values.

4 USER INTERFACE

Having explained the backend implementation, it is important to have an idea of how ThreadViz works in practice. This section summarizes the available tool and visualization features.

4.1 Stepping Forward

Before each step, a thread must be selected in the navigation bar (see Figure 3). By default, the main thread is selected. Once other threads start running, the list of selectable threads expands and each thread is assigned a unique color. In the code editor, the current line is highlighted with the associated thread's pastel color.

For instance, on the left side of Figure 4, the user just performed a step for the selected purple thread, resulting in the current code line changing (highlighted in red). On the right side of the figure, a different orange thread is selected without performing another step. Therefore, its current line remains unchanged.

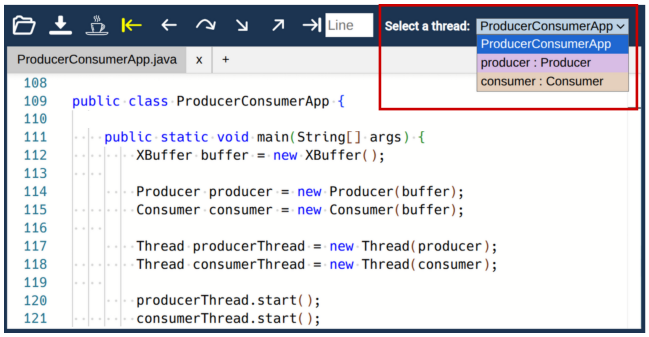

Figure 3: Thread Selection (highlighted in red) in ThreadViz.

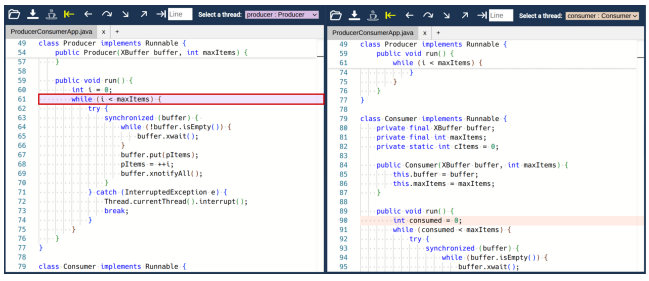

Figure 4: Stepwise Forward Stepping in ThreadViz.

As soon as the current thread blocks due to operations such as `join()` or `sleep()`, the user is forced to continue with another thread. Figure 5 shows the overlay element that appears in such a situation. The user is informed about the current thread's blocking state and must select one of the listed threads to continue. After selecting a new thread, the thread selection in the navigation bar automatically changes to the new thread and the current code line updates accordingly.

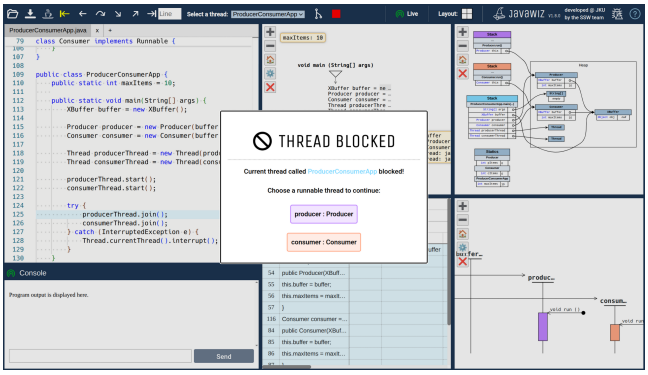

Figure 5: Blocking Notification in ThreadViz.

4.2 Stepping Backward

While stepping back in time, i.e., playing back previous states from the execution trace, the user cannot select new threads nor manipulate already made choices. To achieve this, the thread selection in the navigation bar is disabled (see Figure 6) while stepping back. This ensures that the step history is not modified.

Figure 6: Thread selection is disabled while stepping back.

In this way, there exists the possibility to step back across different thread selections made in the past. According to the step history, the disabled thread selection and current code line are automatically updated. Figure 7 shows an example in which the current code line is at the beginning of the selected thread's `run()` method. When the user decides to step back, the disabled selection changes to the previous thread and the code line is updated accordingly.

4.3 Visualizations

In this section, the enhanced visualizations, including the flowchart view, the memory view, the tabular view as well as the sequence diagram view, are presented that now incorporate useful thread information.

Flowchart View. In ThreadViz, users have access to a flowchart for each thread. Figure 8 shows the setting that enables users to select which threads should be available in the component. As soon as more than one thread is selected, arrows appear on each side of the component, allowing the user to switch between threads.

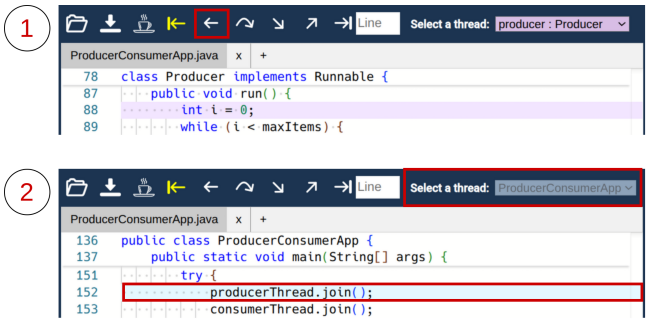

Figure 7: Disabled thread selection and current code line are updated according to thread selection history (both highlighted in red in (2)) when stepping back.

Additional thread information is provided in the right corner of the view which includes both the thread name as well as its current status. Inside each flowchart, one can step by using the step buttons next to the current line. These step buttons disappear when threads block and cannot continue. Moreover, it is possible to have multiple distinct flowchart views open at the same time.

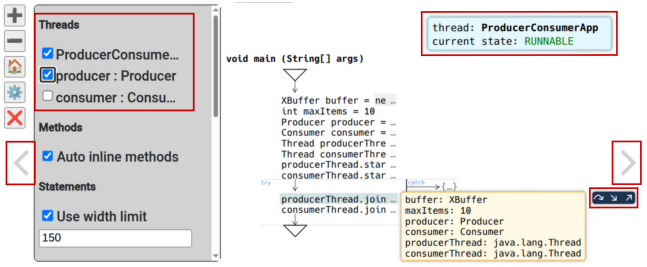

Figure 8: Multiple Flowcharts in ThreadViz.

Memory View. As shown in Figure 9, there now exists a separate stack for each thread, while the heap remains shared. Each stack header is colored in the thread's assigned color.

Tabular View. JavaWiz's tabular view shows how the content of variables changes over time. In the new version (see Figure 10), changing variables and conditions of *all* active methods are visualized, incorporating colors to relate lines to their executing threads. Hovering over elements causes the colors to intensify. To improve readability, the variables and conditions associated with the main thread have been omitted from Figure 10.

Sequence Diagram View. Figure 11 shows an example of a sequence diagram in ThreadViz. Each box is filled with its associated thread's color. Starting another thread is visualized with an arrow that is labeled with `void run ()`. The dot on the right side of the arrow reflects the starting point of the associated thread. It points to the box that represents the lifespan of the thread.

Most importantly, the sequence diagram also visualizes synchronization via locks as well as blocking thread states. Successfully acquiring a lock is represented with a red closed lock icon that has the object label for which the lock was acquired written next to it. The object label has the same color as the thread that holds the lock.

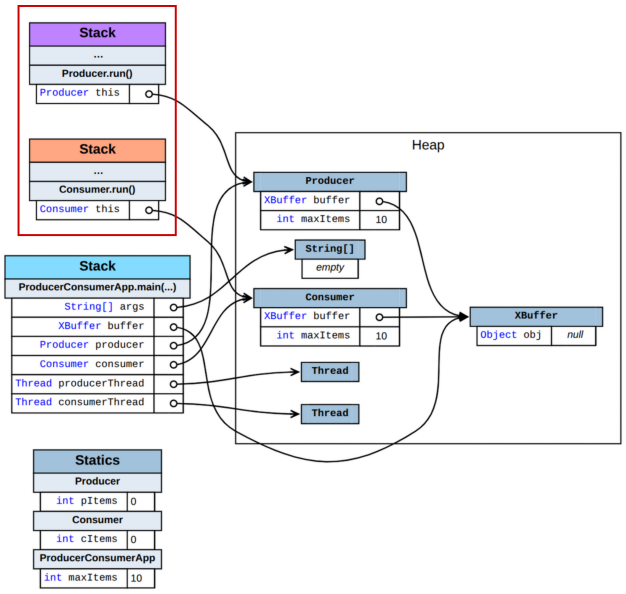

Figure 9: Memory View in ThreadViz.

		Consumer.run()		Producer.run()		
		variables	conditions	variables	conditions	
Line	Instruction	consumed	(consumed < maxItems)	i	(i < maxItems)	(!buffer.isEmpty())
144	consumerThread.start();					
88	int i = 0;			0		
89	while (i < maxItems) {				true	
91	synchronized (buffer) {					
92	while (!buffer.isEmpty()) {					
50	return obj == null;					
92	while (!buffer.isEmpty()) {					false
117	int consumed = 0;	0				
118	while (consumed < ma...		true			

Figure 10: Tabular View in ThreadViz.

Similarly, releasing a lock is shown with a green open lock icon. Waiting or blocked threads are emphasized with gray diagonal lines on top of the related activation boxes.

5 USAGE

To evaluate the capabilities of ThreadViz, six code examples used in the *Internship in Software Development 2* course at the *Johannes Kepler University Linz* are considered. They cover the following scenarios and highlight the benefits of using the tool during teaching:

- *Scenario 1:* Explaining `synchronized`-regions
- *Scenario 2:* Explaining thread pools
- *Scenario 3:* Explaining Java `CountDownLatch`
- *Scenario 4:* Explaining Java `BlockingQueue`
- *Scenario 5:* Explaining Java `ReentrantLock`
- *Scenario 6:* Explaining the Fork/Join framework

In the following, the third scenario is explained to emphasize the tool's educational impact.

Explaining Java `CountDownLatch`. In general, `CountDownLatch` [5] stores a value and implements signal-based coordination via the methods `countDown()` (reducing the value by one) and `await()` (blocking execution until the value reaches zero). Now, let us assume the main execution thread creates three distinct tasks, executed by separate threads, and uses two `CountDownLatches`. There is a `startGate` with `count=1` and an `endGate` with `count=3`.

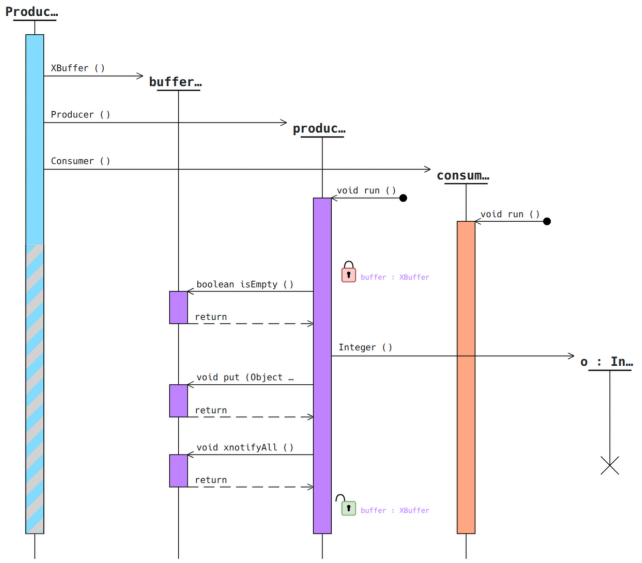

Figure 11: Sequence Diagram in ThreadViz.

To outline the behavior of both `countDown()` and `await()`, the sequence diagram in Figure 12 is generated. The two sections (highlighted in red) are explained to the students in the following:

1. Each thread (violet, orange, and green) tries stepping over `startGate.await()` for each task. As a result, gray diagonal lines appear on top of the related boxes which indicate blocking states. We know that initially the `startGate` has `count=1` which is why `startGate.await()` blocks. When `startGate.countDown()` is called, the `count` is reduced and the waiting tasks are released. Consequently, the gray diagonal lines end shortly after the method call.
2. When `endGate.await()` is executed by the main thread (blue), it blocks. Similarly to before, `endGate` started with `count=3` and the main thread is only released after the `CountDownLatch` becomes `0` (due to the three tasks each calling `countDown()`, signalling that they finished).

After finishing a task, the executing threads terminate. In the sequence diagram, terminated threads are indicated by limiting the height of the boxes that belong to the individual `run()` methods. Additionally, the flowchart view can be shown during teaching to provide more explicit information regarding thread states.

6 LIMITATIONS

Although the presented enhancements contribute significant improvements in terms of supporting novices with understanding different multi-threading concepts, there also exist certain limitations that are worth mentioning:

- *Timeout-approach for identifying blocking states:* As soon as the system load increases, it can happen that a thread is

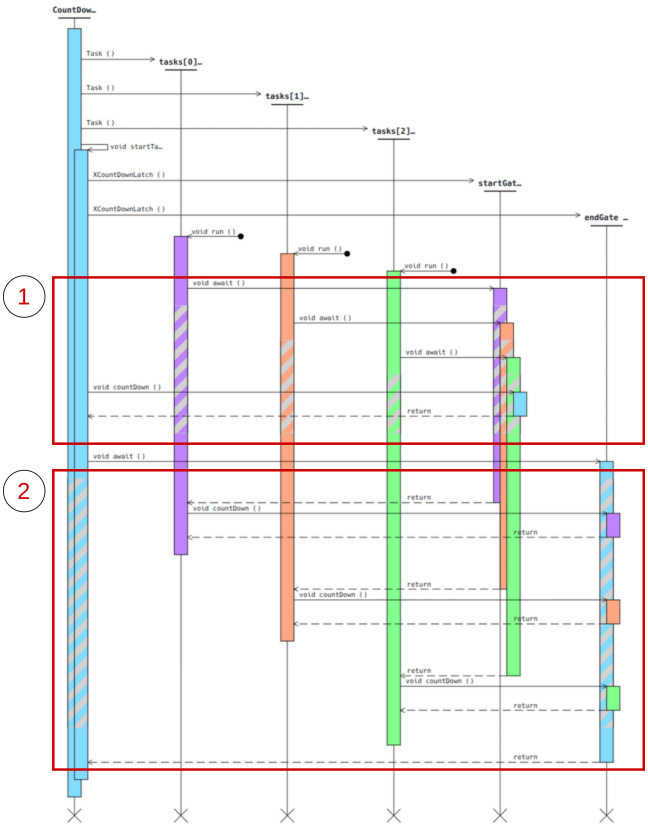

Figure 12: Example for Teaching Java `CountDownLatch`.

identified as blocked although it is actually not. Requests suddenly take longer to be processed. Thus, the timeout might become too short to actually capture the real blocking behavior.

- *Tracing of Java standard library calls:* In Espresso VM, certain *Java standard library* [8] calls such as `wait()` or `notifyAll()` are not traced since Espresso does not suspend native or substituted methods. To include these method calls in the visualizations, code examples need to be rewritten to use encapsulation via inheritance or wrapper classes.
- `Thread.sleep()` *behavior:* So far, sleeping and blocking are both handled the same way. An idea for further improvement would be to simulate the actual behavior by allowing real concurrency.

7 CONCLUSION

This paper introduced ThreadViz, an extension of the trace-based graphical Java debugger JavaWiz with support for visualizing and debugging multi-threaded Java programs. As concurrent execution presents a more sophisticated programming concept that novices often struggle with due to non-deterministic behavior, ThreadViz addresses this by replacing unpredictable concurrency with user-controlled, stepwise thread scheduling.

By allowing users to take over the role of the Java scheduler, each debugging step is associated with the execution of a single thread. This was achieved by utilizing the Truffle Debug API and recording relevant thread information. Based on the collected thread information, the user interface was extended to distinguish between different thread executions. The flowchart view, the memory view, the tabular view as well as the sequence diagram view were adjusted to distinguish between different threads, providing component instances for each thread and incorporating unique colors. Most importantly, the tool provides cohesive visualizations of complex mechanisms such as synchronization, locking and blocking states.

Furthermore, example applications were developed as teaching material to show specific programming scenarios using JavaWiz. The `CountDownLatch` example was discussed in more detail to emphasize the tool's educational applicability and highlight the large scope of possibilities for using it in practice.

Moving forward, we plan to migrate the presented enhancements from the Espresso VM backend to the JDI-based implementation. In this way, they can be made available to end users in the open-source version and support students in learning concurrent programming, as well as teachers in improving the quality of their lectures.

REFERENCES

[1] Alexander Do. *Capture and Replay of JavaWiz Applications in GraalVM Espresso.* Master's Thesis, Johannes Kepler University Linz, Austria, 2025.
[2] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages*, pages 78–90, 2015.
[3] Katrin Kern. *JavaWiz - A Visualization Tool for Software Development Education.* Master's Thesis, Johannes Kepler University Linz, Austria, 2023.
[4] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification.* Addison-wesley, 2013.
[5] Oracle. Class CountDownLatch. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html, 2014. Accessed: 2026-05-19.
[6] Oracle. Enum Thread.State. https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.State.html, 2014. Accessed: 2026-05-29.
[7] Oracle. Java Debug Interface (JDI) — Java™ Platform, Standard Edition 8 Documentation. https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/, 2014. Accessed: 2026-05-19.
[8] Oracle. Java™ Platform, Standard Edition 8, API Specification. https://docs.oracle.com/javase/8/docs/api/, 2014. Accessed: 2026-05-21.
[9] Oracle. Class SuspendedEvent. https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/debug/SuspendedEvent.html, 2026. Accessed: 2026-05-30.
[10] Oracle. Espresso. https://www.graalvm.org/latest/reference-manual/espresso/, 2026. Accessed: 2026-05-10.
[11] Oracle. Introduction to GraalVM. https://www.graalvm.org/latest/introduction/, 2026. Accessed: 2026-05-19.
[12] Oracle. Oracle Labs. https://labs.oracle.com, 2026. Accessed: 2026-05-19.
[13] Oracle. Package com.oracle.truffle.api.debug. https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/debug/package-summary.html, 2026. Accessed: 2026-05-11.
[14] QuestDB. Thread Scheduling. https://questdb.com/glossary/thread-scheduling/, 2026. Accessed: 2026-05-29.
[15] Felix Schenk. *Realization of a JavaWiz backend system running on a JavaScript runtime.* Master's Thesis, Johannes Kepler University Linz, Austria, 2024.
[16] Markus Weninger, Simon Gruenbacher, and Herbert Praehofer. JavaWiz: A Trace-Based Graphical Debugger for Software Development Education . In *2025 IEEE/ACM 33rd Int'l Conf. on Program Comprehension (ICPC)*, pages 1–12, April 2025.