

Author  
**Sonja My Duyen Cao**

Submission  
**Institute for System  
Software**

Thesis Supervisor  
**Dipl.-Ing. Dr. Markus  
Weninger, BSc**

August 2024

# **Synchronization of Hover Effects Across Multiple Visualizations in JavaWiz**



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

In the Bachelor's Program

Informatik



Bachelor's Thesis

**Synchronization of Hover Effects  
Across Multiple Visualizations in JavaWiz**

**Dipl.-Ing. Dr. Markus Weninger, BSc**

Institute for System Software

T +43-732-2468-4361

markus.weninger@jku.at

Student: Sonja Cao

Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc

Start date: March 2024

---

JavaWiz is a teaching tool for software development in Java that visually depicts all steps taken by a program during its execution. It provides various views to help students understand different aspects of the program's behavior, such as the heap-and-stack view, which shows the state of the heap and stack at each step, the list view, which displays the evolution of linked lists, and the array view, which illustrates operations performed on arrays.

One challenge in JavaWiz is that information about a particular entity, such as a list node, can be seen in multiple views simultaneously. For example, a list node may be visible in both the heap view and the list view. To enhance the user experience and facilitate better understanding of the relationships between entities across different views, advanced hover features can be implemented.

The goal of this bachelor's thesis is to develop a synchronized hover effect mechanism in JavaWiz. The main objectives are as follows:

1. Familiarization with JavaWiz's codebase:

- Study and understand the technologies used in JavaWiz, including TypeScript, Vue.js, and D3.js.
- Analyze the existing visualizations, such as the heap-and-stack view, list view, and array view, to gain insights into their structure and functionality.

2. Implementation of advanced hover features:

- Develop a mechanism to detect the type of entity being hovered, such as heap objects, local variables, or class names.
- Establish a communication channel to propagate hover information to all relevant views.
- Update each view accordingly when an entity is hovered, ensuring a consistent and synchronized hover effect across the visualizations.

3. Adoption of at least 3 views:

- Implement the hover synchronization mechanism for the DeskTest view, HeapStack view, and LinkedList view.
- Ensure that hovering over an entity in one view highlights the same entity in other relevant views.
- For example, when hovering over a field of a heap object, the field should be highlighted in all views where it appears, and the object pointed by the field should also be highlighted.

The implementation of synchronized hover effects in JavaWiz will enhance the user experience and facilitate a better understanding of the relationships between entities across different visualizations. By highlighting related entities in multiple views simultaneously, students can gain a more comprehensive understanding of the program's behavior and the interactions between various components.

Modalities:

The progress of the project should be discussed at least every four weeks with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 15.09.2024.



## Abstract

When analyzing Java code it is not always clear how various classes, variables, objects, etc. relate to each other. In particular, novice programmers struggle to understand how their code works, resulting in many unresolved questions. Mapping abstract concepts to concrete implementations can be very complex. Therefore, visualizations, such as flowcharts, desk tests, linked lists, and many more offer to create a better understanding of the control flow.

One tool that provides such dynamic visualizations is JavaWiz. It allows users to understand how Java code is executed and how the program behaves according to the execution. However, even when looking at those graphs it is not always obvious which variable references which object, for instance. This is where hover highlighting comes into play. This feature allows programmers to focus on important information at first glance instead of searching for it first. In many cases, the same information appears in multiple representations, possibly in slightly varying contexts. Therefore, synchronizing the hover between different diagrams proves to be more efficient.

This thesis presents the implementation of such a hover synchronizer in four visualizations: the Tabular Statement History View, the Memory View, the Linked List View, and the Binary Tree View. It creates a cohesive and improved interactive user experience. The synchronization and highlighting of the same information in one or more components should help to quickly identify interesting details and lead to a better understanding of the relationships and interactions within the code.

## Kurzfassung

Bei der Analyse von Java-Code ist nicht immer klar, wie Klassen, Variablen, Objekte, etc. zusammenhängen. Vor allem Programmieranfänger haben oft Schwierigkeiten dabei, die Funktionsweise ihres Codes zu verstehen. Dies führt zu vielen offenen Fragen, auf die sie nicht immer eine Antwort finden. Die Abbildung abstrakter Konzepte auf konkrete Implementierungen kann sehr komplex sein. Daher bieten Visualisierungen wie Flussdiagramme, Schreibtischtests, verknüpfte Listen und vieles mehr die Möglichkeit, den Programm-Kontrollfluss besser zu verstehen.

Ein Tool, das solche dynamische Visualisierungen bereitstellt, ist JavaWiz. Es ermöglicht Benutzern zu verstehen, wie Java-Code ausgeführt wird und wie sich das Programm während der Ausführung verhält. Allerdings ist durch diese Darstellungen nicht immer klar, welche Variable beispielsweise auf welches Objekt verweist. Hier kommt das Hover-Highlighting ins Spiel. Mit dieser Funktion können Programmierer ihren Fokus bereits auf den ersten Blick auf wichtige Informationen legen, anstatt diese zuerst suchen zu müssen. In vielen Fällen enthalten mehrere Darstellungen dieselben Informationen. Deshalb erweist sich eine Synchronisierung des Hovers zwischen verschiedenen Komponenten als äußerst effizient.

In dieser Arbeit wird die Implementierung einer Hover-Synchronisation vorgestellt. Diese wurde in vier Visualisierungen integriert: im Tabular Statement History View, im Memory View, im Linked List View und im Binary Tree View. Sie schafft eine einheitliche und verbesserte interaktive Benutzererfahrung. Die Synchronisierung und Hervorhebung derselben Informationen in einer oder mehreren Komponenten soll bei der schnellen Identifizierung von Zusammenhängen helfen und zu einem besseren Verständnis der Beziehungen und Interaktionen innerhalb des Codes führen.



# Contents

<b>Abstract</b>	<b>i</b>
<b>Kurzfassung</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 JavaWiz Overview . . . . .	3
2.2 TraceState and Retrievable Information from Programming Constructs . . . . .	3
<b>3 Overview</b>	<b>8</b>
3.1 User Interactions in JavaWiz . . . . .	9
3.2 General Idea: Synchronizer for User Interactions . . . . .	10
<b>4 HoverSynchronizer</b>	<b>11</b>
4.1 HoverInfo . . . . .	11
4.2 How Developers can Implement the HoverSynchronizer . . . . .	15
<b>5 Hover Synchronization in the Tabular Statement History View</b>	<b>16</b>
5.1 Detecting Hover . . . . .	16
5.2 Reacting to HoverSynchronizer . . . . .	19
5.3 Providing Information when Hovering Primitive Values . . . . .	21
<b>6 Hover Synchronization in the Memory View</b>	<b>22</b>
6.1 Detecting Hover . . . . .	22
6.2 Reacting to HoverSynchronizer . . . . .	24
6.3 Problems and Solutions . . . . .	25
<b>7 Hover Synchronization in the Linked List and Binary Tree View</b>	<b>28</b>
7.1 Detecting Hover . . . . .	28
7.2 Reacting to HoverSynchronizer . . . . .	29
<b>8 Usage</b>	<b>32</b>
8.1 Constructor and <code>this</code> Reference . . . . .	33
8.2 Hovering Lines and Columns in the Tabular Statement History View . . . . .	35
<b>9 Related Work</b>	<b>37</b>
<b>10 Limitations and Future Work</b>	<b>39</b>
10.1 Limitations . . . . .	39
10.2 Future Work . . . . .	39
<b>11 Conclusions</b>	<b>41</b>
<b>Literature</b>	<b>43</b>





# 1 Introduction

In recent decades, educational software tools have evolved due to technological advances and changing educational needs. Initially used mainly in adult training or higher education [18], these tools are now employed at nearly all educational levels.

In software engineering education, these tools assist both teachers and students. Learning software topics can be challenging, especially for first-year students who struggle with understanding basic concepts [12]. Therefore, educational tools promote interactive and collaborative learning, aiding students in understanding complex concepts rather than just memorizing facts [17].

In terms of programming, learning a programming language is similar to acquiring a foreign language. However, instead of applying known practices, a programming language requires a more structured approach. Visualization tools have gained popularity for simplifying complex concepts through illustrations. They help students grasp abstract ideas, engage in observing code behavior, and provide immediate feedback, which boosts the students' motivation.

Tools like BlueJ [13], Greenfoot [11], and Scratch [6] offer significant benefits but also have limitations. For instance, Greenfoot's playful approach to teaching object-oriented programming isn't always suitable for higher education. Here the focus lies on conveying material rather than teaching by "playing around". Moreover, many tools provide one visualization only. This is where JavaWiz comes in.

JavaWiz is a visualization tool developed by the Institute for System Software at the Johannes Kepler University in Linz [7]. It assists people new to programming by visualizing the execution of Java code. Some components that are already in use are:

- A code editor with a debugger
- A flowchart diagram
- A memory visualization
- A statement history visualization as a table
- A terminal for inputs and outputs
- An array visualization
- A linked list visualization
- A binary tree visualization

This educational tool is available not only to educators and students at JKU but also to anyone interested in learning Java visually. JavaWiz is a web application that can also be installed as a Visual Studio Code plugin. Even though, JavaWiz has proven to be beneficial to beginners, various visualizations do not display visually emphasized relations between components to the user.

In the context of this thesis, JavaWiz is extended by a hover highlighting feature. The so-called "HoverSynchronizer" ensures a consistent user experience by linking interactive elements across visualizations. This work covers the implementation of the HoverSynchronizer in four supported illustrations: the Tabular Statement History View, the Memory View, the Linked List View, and the Binary Tree View.



## 2 Background

This section outlines the motivation behind the creation of the educational visualization tool JavaWiz, providing a deeper understanding of the implementation of certain features. This also includes a brief introduction to the teaching methods used in programming classes at JKU and the online course in which the tool will be utilized.

### 2.1 JavaWiz Overview

The visualization tool JavaWiz was first made publicly available by the Institute for System Software at the Johannes Kepler University in Linz in 2022 [10]. Its primary objective is to aid novice programmers in their learning process by offering various visualizations that illustrate the execution of Java code. This makes abstract concepts more accessible and programming structures easier to grasp. However, this tool is not only meant to be used by beginners but also by educators. It serves as a valuable resource for educators, who can integrate its features into their teaching materials to teach programming principles more effectively. This includes reference semantics, the program flow, such as loops in combination with arrays, or operations on linked lists.

### 2.2 TraceState and Retrievable Information from Programming Constructs

JavaWiz provides a variety of graphs for different aspects in software education, which will be further explained in Section 3. In general, all features in JavaWiz visualize a collection of states of a program in execution. Those are referred to as trace states represented by the `TraceState` interface. The following information can be retrieved from a trace state:

- **sourceFileUri:** The name of the Java file of which the current code is executed.
- **line:** The currently active and processed line in the source code.
- **stack:** The stack contains all function calls that are currently active. Each function is stored as a stack frame and contains the method signature, parameters, local variables, and the Java class that contains the method and conditions of e.g. if-else-statements.
- **heap:** The heap is responsible for dynamic memory allocation. It contains all objects that are created during runtime. When an object is not referenced anymore, the so-called garbage collector terminates the object and frees up space.
- **loadedClasses:** A loaded class is a class already loaded during runtime to the Java Virtual Machine. It contains all static fields along with the class name.
- **output, error, and input:** Those three contain `stdout`, `stderr`, and `stdin` that are produced after the execution of a code line.
- **inputBufferInfo:** This field describes what input in the input buffer has already been processed, what still needs to be processed, and what has just been processed at a specific state.

The gathered information can further be grouped into the following categories:

### **Code Lines**

Code lines are uniquely identified by their line number and file URI. The URI, short for Uniform Resource Identifier, specifies to which file the currently executed code belongs. It usually consists of the path to the file, including the filename and prefixed with `file://`.

### **Classes**

Classes are referred to as `LoadedClass` in JavaWiz and are identified by their name and static fields. In Java, it is possible to create two or more classes with the same name. Therefore, the URI which is stored in a trace state differentiates those from another.

### **Methods**

A stack frame contains all relevant information about a method, such as a method name, signature, local variables, conditions of loops and statements, etc. A method is uniquely identified by its name along with the class from which it is called.

### **Conditions**

Conditions consist of an expression that returns a truth value. They appear in if-else statements, switch statements, and loops. Often the same condition is checked more than once during runtime. Therefore, each condition is assigned an ID to differentiate those checks in time. For instance, the expression `a < b`, returns `true` if the condition is truthy and `false` otherwise.

### **Local Variables**

Local variables are uniquely identified by their name additionally to the class and method in which they are accessed. Furthermore, each local variable can store a primitive value, a reference to a heap object, or `null` as its value. They are stored on the stack. In particular, local variables are stored in the stack frame of a method.

### **Static Fields**

Similar to local variables, static fields are identified by their name and class name. Each static field can store a primitive value, a reference to a heap object, or `null` as its value. In order to retrieve the information about a static field, the static fields of a loaded class are accessed.

### **this**

The current instance of a class is referenced by the keyword `this`. It is handled differently than local variables or static fields and contains only the reference to the class instance. The handled a little differently compared to local variables. Additionally, an identifier is assigned.

### **Heap Objects**

Every time an object is created, it is allocated in the heap. The heap object can then be accessed with its unique ID, which is assigned during the allocation. This identifier is also called "reference" to the space in which the object is allocated. Additionally, the kind of an object is stored. JavaWiz specifically differentiates between arrays, strings and other objects.

## Object Fields

Similar to local variables and static fields, the fields of an object have a name. Each field can store a primitive value, a reference to a heap object, or `null` as its value. However, the difference is that instead of the class name and method name, the ID of the object determines the uniqueness of the field. When an object is created, it and its fields are stored in the same interface.

## Array Cells

In Java, an array receives an allocated space in the heap. Therefore, it is handled as a heap object in JavaWiz. In addition to the object ID, the distinction between cells in an array is achieved by storing the corresponding index at which it can be accessed. Furthermore, the value of each cell is stored as well. This can be a primitive value, a reference to a heap object of `null`.

By understanding these categories, programming beginners and students can gain a deeper insight into the state of a program during its execution. Furthermore, each category is further identified by their kind, such as "Local" for local variables or "HeapObject" for allocated objects in the heap. Complex behaviors are represented in such a way to be easier understood. This is particularly beneficial for educational purposes, as it simplifies the learning process.

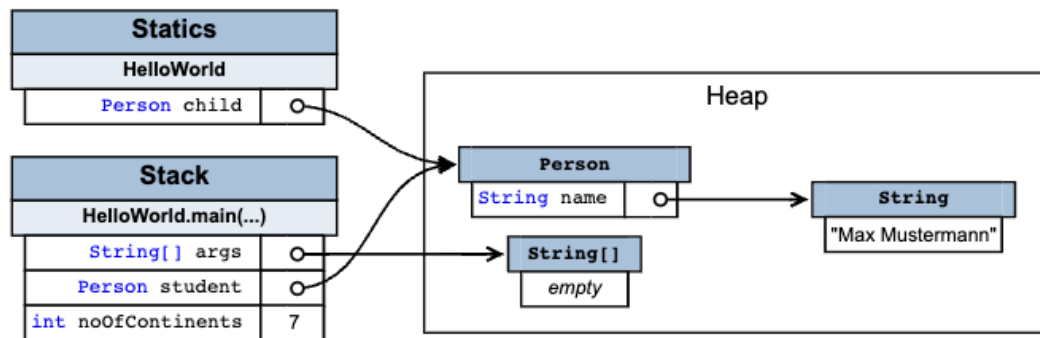


Figure 1: Example of a local variable and static field referencing the same heap object.

Figure 1 visualizes the stack and heap of a simple Java code. In the example above `HelloWorld` and `Person` are loaded classes since they are defined by the programmer. Moreover, the class `HelloWorld` has a method named `main` for which a stackframe is created.

Besides the default `args` array of the main method, the heap contains two other objects with the structure:

Object ID, Type

-----

#1, "Person"

#2, "String"

The class `HelloWorld` has a static field `child` that references the `Person` object on the heap. That object has a field `name` that references the `String` object, which has the primitive value "Max Mustermann". Furthermore, the method `main` creates a local variable `student` on the stack. It can be seen that this local variable references the same object as the static field. In the

same method another local variable `noOfContinents` is created which stores a primitive integer value.

Based on this information the following structure can be identified:

```
Kind, [Object ID for fields,] Name, Value
-----
"Static", "child", #1
"Local", "student", #1
"Field", #1, "name", #2
"Local", "noOfContinents", 7
```

The information about a local variable can be obtained by accessing it from the stackframe of the method where it is created. On the other hand data about a static field is retrieved from its loaded class. In contrast to those two kinds, information about a heap object can be found on the heap with the object's identifier. Listing 1 shows how such a data access could look like based on the previous examples.

Listing 1: How to access a local variable, static field or heap object

```
// Accessing a local variable
traceState.stack[0].localVariables.get("noOfContinents");

// Accessing a static field
traceState.loadedClasses[0].staticFields.get("child");

// Accessing a field of an object
traceState.heap.get(1);
```

It is noteworthy, that the index 0 when accessing the local variable in Listing 1 indicates the main method. Likewise, when a static field is accessed, the index 0 refers to the class `HelloWorld` in this example. This is because the class is the first one that was loaded. The class `Person` would have the index 1 since it is the second loaded class. Similar to those two examples, a heap object is retrieved by its ID, which is an object of type `Person` in this case.



### 3 Overview

In general, JavaWiz consists of a backend written in Kotlin and a frontend application developed with written with the framework Vue.js. The backend introduces a compiler and debugger while the frontend visualizes the data output sent by the backend. The debugger allows the user to execute the program line-by-line. As a result, many different states during code execution are visualized and analyzed.

The visualization tool can be run locally as a web application in a browser, as seen in Figure 2, or as a Visual Studio Code extension.

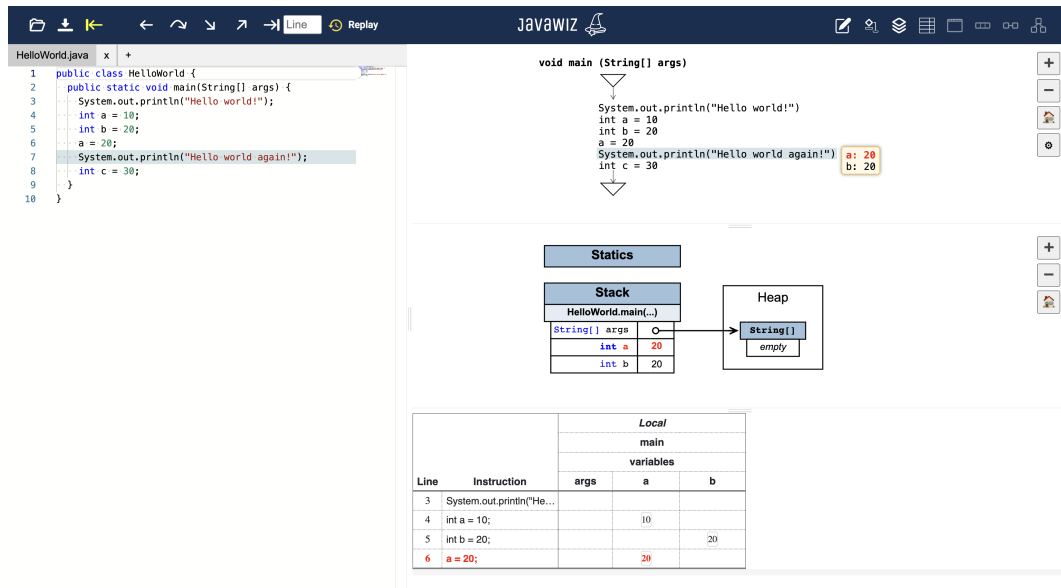


Figure 2: JavaWiz web application.

Figure 2 shows one possible use case of the JavaWiz web application. Users are given the opportunity to either write their own Java code in the editor on the left panel or upload an existing file or project. The program can then be debugged line-by-line.

There are numerous open source code editors available that provide a variety of features. Therefore, JavaWiz utilizes the Monaco Editor by Microsoft. Among other things, it offers syntax highlighting, supports multiple programming languages, such as Java, and introduces a model that collects data to be further processed [16].

In JavaWiz the majority of the graphs are drawn with "d3-graphviz" [9]. d3-graphviz utilizes features from both D3.js [1] and Graphviz [8]. The JavaScript library D3.js (Data-Driven Documents) allows the creation of dynamic and interactive graph elements. Graphviz is a visualization software that renders graphs written in the DOT language and automatically adjusts a graph's layout.

JavaWiz offers a variety of visualizations, which are briefly explained in the following:

- **Flowchart:** This type of diagram shows each step in the sequential execution of a process or algorithm. It describes how decisions are made and the results of an action. Its purpose



is to illustrate how the process or algorithm works from start to finish.

- **Memory:** In Java, stack and heap are two different areas where data is stored. Local variables, that are used within a method, are temporarily stored on the stack. The heap mostly comprises objects, particularly data that is stored permanently after a method has been executed until they are not needed anymore. Additionally, this component displays static variables besides local variables and objects.
- **Tabular Statement History:** The Tabular Statement History View behaves like a desk check, which is usually done with pen and paper. With the check, the programmer has to run through a Java code manually. This should help to detect and eliminate errors promptly. In JavaWiz this is visualized as a table containing methods and variables as column headers and code lines as row headers. According to the executed line, the table shows what value a variable has at the time.
- **Console:** The console serves as a visualization for the input and output behavior visible to the user. The client can enter something if asked to do so by the program. They also see the output of e.g. a line that prints a message with `System.out.println`.
- **Arrays:** This visualization is used to demonstrate how data is processed and stored in an array.
- **Linked List:** Each element is called a node that typically contains a value and a reference to the next node. Multiple nodes can be connected by setting the reference. As a result, a chain-like structure is produced.
- **Binary Tree:** A binary tree is similar to a linked list. The difference is that it contains two references that are usually labeled as the left or right subtrees and leaves. It is very popular to visualize and teach specific algorithms such as searching or sorting.

### 3.1 User Interactions in JavaWiz

JavaWiz not only allows the creation and modification of dynamic visualization but also offers a variety of interactive elements. This allows novice programmers to study programming principles at their speed.

As already mentioned the navigation bar allows users to toggle between visualizations. Additionally, it is possible to execute the same steps as a debugger. More precisely, users can control the speed of how the Java code is executed by using the step buttons in the top left corner. This includes "step back", "step over", "step into" and "step out".

Furthermore, many visualizations created with d3-graphviz allow for the expansion or collapsing of specific elements. For instance, variables or fields that reference an object can be collapsed or expanded to hide or show that object.

Additionally, those illustrations provide a couple of controls on the right side of the corresponding component. With those programmers can zoom in and out of visualizations, center the view within the component, and specify custom settings if available.

In contrast, to the individual interactions that users see, the underlying data structure of JavaWiz

is nearly the same for every component. Depending on what should be visualized, the essential information is retrieved from trace states. How the data is processed depends on the elements to be rendered. Some may be visualized as tables and others represent a structure composed of nodes of different shapes and edges.

Even though JavaWiz already provides numerous ways for users to interact with the visualization tool, the interactions are mainly focused on one component only. Sometimes users would like to see how multiple components relate to each other. For instance, which object is created or referenced when a line in the Tabular Statement History View is executed. This requires the implementation of a feature that allows the synchronization of such an interaction.

### 3.2 General Idea: Synchronizer for User Interactions

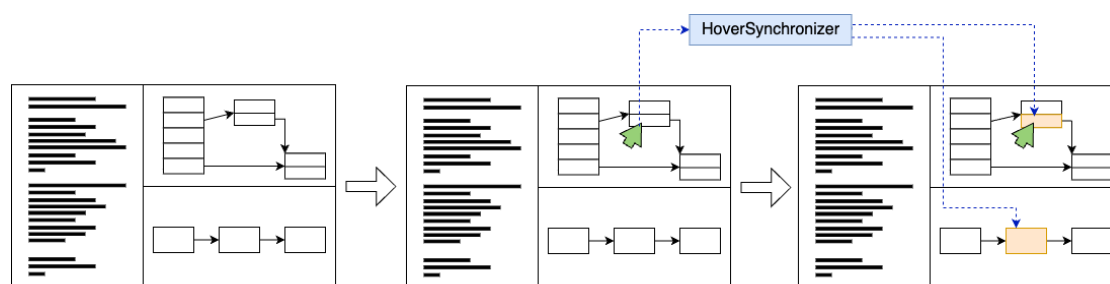


Figure 3: Idea of a synchronized hover in JavaWiz.

Figure 3 shows how introducing a synchronized feature could look like. In particular, the figure shows the implementation of a synchronized highlighting of the same elements across features. As of the current JavaWiz release, no such feature is available. Therefore, this thesis introduces the proposed idea by reacting to hover interactions, which are triggered by JavaScript mouse events.

As soon as the user hovers over an element, such as the node value of a linked list, the same element that is illustrated in another component, like the field of the object in the heap, is highlighted, too.

In the scope of this thesis, four already utilized visualizations were modified. Namely the Tabular Statement History View, the Linked List View, the Binary Tree View, and the Memory View. The following sections further elaborate the implementation details of the hover synchronization.

## 4 HoverSynchronizer

The class `HoverSynchronizer`, see Figure 4, is responsible for synchronizing data between registered components. Before the hovered information is sent, it has to be prepared beforehand, see Section 4.1. For performance and efficiency reasons the data structure is consistent for all components. The only differences lie in how each component processes the received data or prepares it for other visualizations. Section 5, Section 6 and Section 7 further explain how this is done in the Tabular Statement History View, Linked List View and Binary Tree View as well as the Memory View.

In general, the `HoverSynchronizer` has a static set that contains the information that is currently hovered in the mentioned visualizations. The class implements similar ideas of the Observer Design Pattern.

A component that wishes to access the synchronized information has to register its handler method by calling the static `onHover` method of the synchronizer class. The newly registered handler is added to the set of listeners. As soon as the user hovers an element in a visualization, the `HoverSynchronizer` updates the hovered data and notifies all listeners. This can be achieved by calling the static `hover` method.

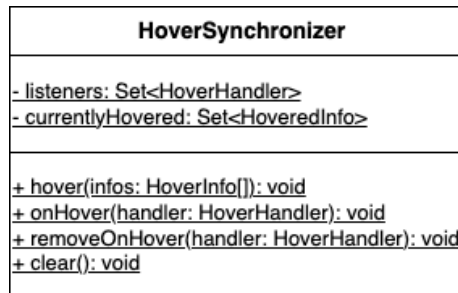


Figure 4: `HoverSynchronizer` class.

### 4.1 HoverInfo

Overall, the main goal is to provide a feature that allows the highlighting of the same elements in different visualizations when hovered by the user. Therefore, a common interface `HoverInfo` was created to ensure a consistent data structure for visualizations now using the new feature and more importantly future implementations in other components.

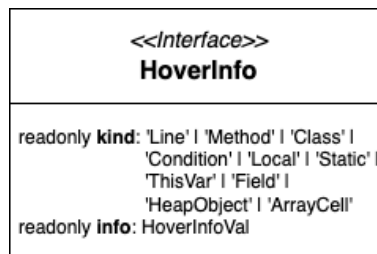


Figure 5: `HoverInfo` interface.

Figure 5 shows the general structure of the `HoverInfo` interface. In total, ten different kinds of information can be hovered. They further simplify the information of a trace state and only contain the information necessary to uniquely identify each type. Since the new feature solely introduces the highlighting of elements, it is not necessary to store further details. Only the determination of the elements to be highlighted is of interest. To create a better understanding of how this is achieved, the information that the synchronized hover provides is explained in the following:

Interface / Type	Description
<pre>HoverLine {     lineNr: number     localUri: string }</pre>	Each line in a Java file has a number. However, mainly storing the line number is not sufficient. JavaWiz allows users to upload or create multiple files. Therefore, the URI to the corresponding file is passed as additional information.
<pre>HoverClass {     class: string }</pre>	If the user hovers over a visual element that represents a Java class, the name of this class is passed as the hovered information. With this other visualizations are able to identify and highlight matching elements if present.
<pre>HoverMethod {     class: string     method: string }</pre>	In general, a method is executed in a specific class. Therefore, the name of the class in which the method is created and the method's name are needed to determine which method is hovered. The reason behind this is that sometimes different classes have methods with the same signature.
<pre>HoverCondition {     class: string     expression: string }</pre>	Visualizations that contain information about the evaluation of conditions of e.g. a if-else statement can pass the expression. To correctly allocate the condition to its origin, the class name is passed as an additional piece of information. In the scope of this thesis, only the Tabular Statement History View contains conditions. Other visualizations might require other or more data to accurately highlight concerning elements, which is briefly explained in Section 10.

Interface / Type	Description
<pre> HoverLocal {   class: string   method: string   name: string   reference: number } </pre>	<p>This type serves as the information about which local variable is hovered. Also, in this case, variables with the same name can exist. As a result, other visualizations are additionally notified about the name of the method that creates the variable. On top of that, the name of the class where the method is created needs to be transferred as well with the same reasoning as in <code>HoverMethod</code>. The property <code>reference</code> contains the reference to an object if the local variable can and actually references one. If this is not the case, the value of this property is <code>-1</code>.</p>
<pre> HoverStatic {   class: string   name: string   reference: number } </pre>	<p>Similar to local variables, the name of the static field is passed as a hover information. The only difference is that in the case of static fields, only the name of the class where they are created is of interest. It has something to do with the fact that they are created within the scope of a Java class and exist during the entire runtime rather than in a method only. Here the reference to an object on the heap can be added to the data again.</p>
<pre> HoverHeapObject {   objId: number } </pre>	<p>A newly created object is uniquely allocated a space on the heap. That space contains an ID with which it can be referenced. Therefore, it is adequate to send the identifier as the hovered information only.</p>
<pre> HoverField {   objId: number   name: string   reference: number } </pre>	<p>At first glance, the structure of the information seems to be a combination of the types <code>HoverHeapObject</code> and <code>HoverLocal/HoverStatic</code>. As mentioned before, the ID of the allocated memory on the heap uniquely identifies an object. To further differentiate between its fields, the name of the hovered field element can be passed along. Also, fields can reference another object which is why said reference might be interesting to know as well.</p>

Interface / Type	Description
<pre>HoverThisVar {   methodId: number   name: string   reference: number }</pre>	<p>This type was created for the purpose of <code>this</code> references only. It refers to the current instance of an object. When a method in that instance is called, the reference points to the current object where this method was invoked. Therefore, the property <code>methodId</code> represents the index to the method's stack frame.</p>
<pre>HoverArrayCell {   objId: number   index: number }</pre>	<p>Section 2.2 already explained how JavaWiz handles cells of an array. A newly created array is allocated a memory space on the heap. To access a specific cell, the index is passed as additional information.</p>

Table 1: Structures of the hovered information shared in multiple visualizations.

Besides the class `HoverSynchronizer` and all related types of the common type `HoverInfo`, the hover highlighting feature introduces helper functions. More specifically, factory methods were created for the purpose of reuse in the current implementation and future applications. As a result, the newly added code remains as short and compact as possible while providing enhanced performance. Furthermore, code smells are reduced. Particularly, centralizing the creation of `HoverInfo` objects makes code maintenance easier.

In the scope of this feature, the introduced factory methods create a `HoverInfo` object. They are responsible for setting the correct properties according to the kind of element being hovered. Developers who are working on JavaWiz are also able to modify the logic behind the creation once for all components instead of having to modify each one individually.

The following code snippet shows how one of the factory methods looks like. It creates an object that represents the cell of a Java array.

```
export function createHoverArrayCell (objId: number, index: number): HoverInfo {
  return {
    kind: 'ArrayCell',
    info: {
      objId,
      index
    } as HoverArrayCell
  }
}
```

The property `kind` provides information about what type of element the user has hovered. `info` is the property for setting the concrete data to differentiate the array cell among other objects and cells.

## 4.2 How Developers can Implement the HoverSynchronizer

In order to implement the hover highlighting feature into a component, there are two separate chores. The first one is to specify the handler method that is registered as a listener of the class `HoverSynchronizer`. The second task is to retrieve the hovered information from the component and prepare it accordingly for other visualizations.

### Specifying the Handler Method

The handler method is responsible for preparing the data it receives from the `HoverSynchronizer` to be further processed in a component. For this, the method has to be registered as a listener by calling `HoverSynchronizer.onHover(myHandler)` which is usually done when the visualization is mounted. Whenever an element in a different illustration is hovered, all listeners are notified about the new hovered information. It is an array of type `HoverInfo` passed as a method parameter. The components that were modified in the scope of this thesis save the hovered information as a part of the object that is returned by their data function. This ensures that the data can be accessed from anywhere within the code. As a matter of fact, not only does every component handle the received information differently but also various methods within one. Therefore, it is challenging to find a uniform solution that fits all and enables reuse.

After the registration of a listener when the visualization is mounted follows the removal of that listener when it is unmounted. The removal is similar to the registration but a different method is called, namely `HoverSynchronizer.removeOnHover(myHandler)`. Otherwise, by closing and opening a visualization, JavaWiz tries to call the handler method from before that does not exist anymore. As a result, the visualization cannot be opened due to thrown errors.

More information about the class `HoverSynchronizer` and all `HoverInfo` types can be found in Section 4.

### Retrieve and Prepare HoverInfo

As noted earlier, each method that is in charge of defining the elements to be drawn, handles the hovered information differently. The functionality introduced by the new feature must be integrated by modifying the existing code rather than creating a separate part. Therefore, it is of utmost importance to first analyze the code before incorporating the feature. Developers need to figure out where the mouse event listeners can be added and to which elements in the visualization. For instance, in the Linked List View the listeners were added to the rectangles of the value field and the next field as well as pointers.

Different JavaScript `MouseEvents` exist, but the hover highlighting feature mostly requires the two events `mouseenter` and `mouseleave`. They take care of preparing the array of type `HoverInfo` that is passed as a parameter for calling `HoverSynchronizer.hover(newHoverInfos)`. Within that static method, the set storing the hovered information is updated and all listeners are notified. Another noteworthy point is that the handler method of the visualization in which the user currently hovers an element is called as well.

Besides specifying and processing the hovered information, each component must determine which element or shape shall be emphasized. Ideally, the modified visualization should consider the chosen color code for the different information types, see Section 6.2. Since this matter might require a more complex logic, the components that were modified already, include methods that help to evaluate and choose the correct color.

## 5 Hover Synchronization in the Tabular Statement History View

As visible in Figure 6, the Tabular Statement History View displays a table imitating a desk check usually done with pen and paper. It shows the control flow of a Java program step-by-step. The most important information perceived by users are lines of code, methods, local variables, static fields, expression of conditions, and assigned values. This section gives an overview of the implementation of the synchronized hover highlighting feature in the Tabular Statement History View.

Line	Instruction	Local							Static		
		main									
		variables							conditions		
		args	a	b	diff	arr	c	c2	a < b	Cat.NR_OF_LEGS	Cat.SIBLING
15	System.out.println("He...										
16	int a = 10;		10								
17	int b = 20;			20							
18	int diff = diff(a, b);										
39	if (number1 < number2) {										
40	return number2 - num...										
18	int diff = diff(a, b);				10						
20	int[] arr = new int[] { 17...					int[]					
22	Cat c = new Cat();								0	null	
2	static int NR_OF_LEG...								4		
3	static Cat SIBLING = n...										
3	static Cat SIBLING = n...										Cat
22	Cat c = new Cat();										
22	Cat c = new Cat();						Cat				
23	Cat c2 = new Cat();										
23	Cat c2 = new Cat();							Cat			
25	while(a < b) {								true		

Figure 6: Tabular Statement History View.

### 5.1 Detecting Hover

A table is two-dimensional which means that information can be retrieved from columns and rows. This is also the case in the Tabular Statement History View. Headers are names of methods, local variables, static fields, and expressions. While the method name solely specifies a single table cell, the latter three are associated with a column. Rows indicate which line of code is currently executed based on the line number and instruction. Therefore, two different use cases were evaluated and considered during the feature implementation.

Since a hover user interaction is a mouse event, fitting types of JavaScript event listeners needed to be added to the hovered elements. In the JavaWiz version that is currently available to all users, the component already utilizes two types, namely "mouseover" and "mouseout". In those, the event handler specifies which lines should be highlighted based on what the user hovers in the Tabular Statement History View. Thus, JavaWiz has a component that reacts to mouse events. However, this is only possible in one visualization.

After evaluating the two event types a conclusion an interesting point was found. Those types caused performance issues and were hence replaced by the "mouseenter" and "mouseleave" events. This issue was not detected in the Tabular Statement History View but later in the Memory View. Section 6.3 explains those findings in detail.



### Hover over a Table Header

Starting from the top, when hovering the name of a method, JavaWiz provides an object containing the information needed to specify the hovered information. As discussed in Section 4.1, a method is identified by its class and name. With this information, a `HoverInfo` object can be created that has the structure as specified for `HoverMethod`.

```
{
  kind: "Method",
  info: {
    class: "HelloWorld",
    method: "main"
  } as HoverMethod
}
```

Listing 2: `HoverMethod` object.

Similar to methods, the visualization tool delivers information about the three remaining entities as well. The structure for local variables and static fields are pretty similar. Variables have four properties which are then accordingly mapped to the structure of `HoverLocal`:

```
{
  kind: "Local",
  info: {
    class: "HelloWorld",
    method: "main",
    name: "a",
    reference: -1
  } as HoverLocal
}
```

Listing 3: `HoverLocal` object.

The hovered information for static fields is comparable to local variables. The only difference is that the information does not contain the property `method` because they are globally accessible in Java. Furthermore, the retrieved data is mapped to the structure of type `HoverStatic`.

When looking closely, the property `reference` has the value `-1`. The reason is that the variable does not point to an object on the heap but to a primitive. For the scenario that it references one, the property would have the object's ID as its value.

Regarding conditions, the structure returned by hovering over the header contains an expression string and the name of the Java class in which a condition is evaluated. Exactly this data is mapped to a new `HoverInfo` object with the structure of the type `HoverCondition`:

```
{
  kind: "Condition",
  info: {
    class: "HelloWorld",
    expression: "a < b"
  } as HoverCondition
}
```

```
}
```

Listing 4: HoverCondition object.

The hovering of table headers solely requires the proper mapping of data to be processed by the synchronization feature. For this reason the method `onHoverHeader` was created to do exactly that. It receives a parameter that has the type `DeskTestMethod`, `DeskTestVar`, `DeskTestStatic`, or `DeskTestCondition`. The object already holds the necessary information for the new feature which is why the mapping to instances of `HoverInfo` is straightforward.

### Hover over a Row in the Table

When hovering over a row in the visualization, JavaWiz provides the following information:

```
{
  line: 17,
  localUri: "HelloWorld.java",
  stateIndex: 2,
  methods: [
    {
      name: "main",
      class: "HelloWorld",
      vars: [
        {
          class: "HelloWorld",
          name: "args",
          method: "main(...)",
          type: "java.lang.String[]"
        },
        {
          class: "HelloWorld",
          name: "a",
          method: "main(...)",
          type: "int"
        },
        {
          class: "HelloWorld",
          name: "b",
          method: "main(...)",
          type: "int"
        }
      ]
    },
    conditions: []
  ],
  statics: [],
  currentVars: {},
}
```

```

        currentConditions: {},
        currentStatics: {}
    }

```

Listing 5: Information received by JavaWiz when hovering a desk test line.

The properties `line` and `localUri` are used to specify a object representing `HoverLine`. Since methods solely occur in the table header, the property `methods` is ignored. Despite that the main focus was on the properties `currentVars`, `currentStatics` and `currentConditions`.

The information that a hovered row returns contains the information available after a code line was executed. In particular, the three properties are always except for when local variables or static fields are created or used and when conditions are evaluated. To map a table row to a structure that can be consumed by the hover highlighting feature, another method was developed.

At first `onHoverLine` creates a `HoverLine` object which is simple. Afterward, the implementation checks if `currentVars`, `currentStatics` or `currentConditions` contain a `DeskTestVar`, `DeskTestStatic` or `DeskTestCondition` object. In case one of them does, the interested data is retrieved with `deskTestLine.currentVars.entries().next.value` and then mapped to the according types supported by `HoverInfo`.

## 5.2 Reacting to HoverSynchronizer

Whenever a visual element is hovered, the method `inHighlightedHeaders` within the Tabular Statement History View is responsible for specifying the correct CSS class for the table's columns. The name of the class that is responsible for setting the emphasized background color is `highlighted-col`. In order to do so, all table cells in the same column can pass the information retrieved from their header as a parameter. Each header data can have the type `DeskTestMethod`, `DeskTestVar`, `DeskTestStatic`, and `DeskTestCondition`. Based on these types, the steps to check whether the column should be highlighted differ. It is noteworthy that the colors for highlighted elements are consistent for all visualizations to ensure clarity.

To properly highlight elements related to the hovered data, the method `inHighlightedHeaders` iterates over all hovered information. Within one iteration the data retrieved from the header is compared to the hovered information. In particular, the values of the same properties are checked. If all according properties are correlated, the method returns "true" and "false" otherwise. With this boolean value, the CSS class is conditionally assigned to the correct table cells to highlight an entire column.

Figure 7 shows the emphasis on a method name. Since the component focuses on the local variables, static fields, and conditions of a method, solely a single table cell is emphasized when hovering a visual element representing a method.

		Local							
		main							
		variables							conditions
Line	Instruction	args	a	b	diff	arr	c	c2	a < b
15	System.out.println("He...								
16	int a = 10;		10						
17	int b = 20;			20					

Figure 7: Tabular Statement History View: Highlighting when a method is hovered.

On the other hand, there are two possibilities to hover and highlight one of the three mentioned programming constructs. The first is the highlighting of a column only, see Figure 8. This can be achieved by either hovering the table header in the Tabular Statement History View or when the construct is hovered in a different component.

		Local								Static	
		main									
		variables							conditions		
Line	Instruction	args	a	b	diff	arr	c	c2	a < b	Cat.NR_OF_LEGS	Cat.SIBLING
15	System.out.println("He...										
16	int a = 10;		10								
17	int b = 20;			20							
18	int diff = diff(a, b);										
39	if (number1 < number2) {										
40	return number2 - num...										
18	int diff = diff(a, b);				10						
20	int[] arr = new int[] { 17...					int[]					
22	Cat c = new Cat();									0	null
2	static int NR_OF_LEG...									4	
3	static Cat SIBLING = n...										
3	static Cat SIBLING = n...										Cat
22	Cat c = new Cat();										
22	Cat c = new Cat();						Cat				
23	Cat c2 = new Cat();										
23	Cat c2 = new Cat();								Cat		
25	while(a < b) {								true		

Figure 8: Tabular Statement History View: Highlighting of a column when a table header is hovered.

The second possibility is the highlighting of a line and the column containing the information retrieved by it which is shown in Figure 9. If the user interacts with the visualization and hovers over a line, all lines representing the hovered line are emphasized. This feature was already in use before the implementation of the synchronized hover. The new feature now additionally highlights the corresponding column to accentuate the programming construct that was created or accessed after executing the hovered line.

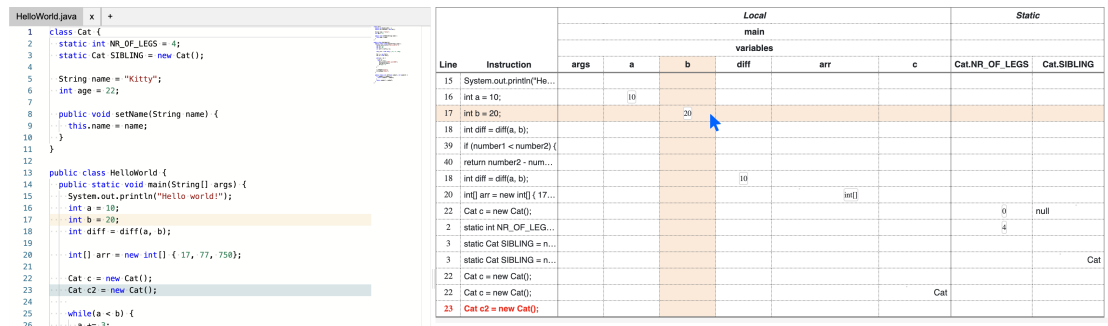


Figure 9: Tabular Statement History View: Highlighting when a line is hovered.

### 5.3 Providing Information when Hovering Primitive Values

Sometimes the table becomes very large and does not entirely fit on the screen. Therefore, when the user would like to know to which variable or static field a value belongs, they have to scroll all the way to the top. While doing so, it is important to focus on the same column to see the correct header. Especially with large tables, this becomes an issue.

The solution introduced by the hover highlighting feature incorporates a reaction to hovering a primitive value, see Figure 10. This user interaction opens an overlay holding important information, including the variable's name, type, and the class in which it is created.

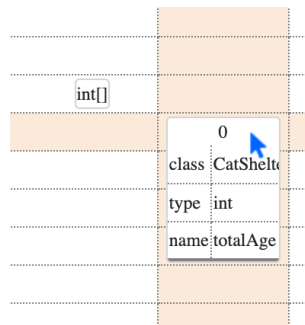


Figure 10: HoverSynchronizer: More information is displayed when a primitive value in the Tabular Statement History View is hovered.

## 6 Hover Synchronization in the Memory View

Initially, JavaWiz draws the stack containing local variables and methods as well as static fields on the left side. The heap contains all created objects and their fields and is displayed on the right side of the graph, see Figure 11. The positioning is automatically handled by the library d3-graphviz, which is further elaborated in Section 3.

In addition to the provided user interactions, that allow users to resize the visualization or collapse and expand object references, the ‘HoverSynchronizer’ was implemented. This section covers the implementation of the synchronized hover highlighting feature in the Memory View.

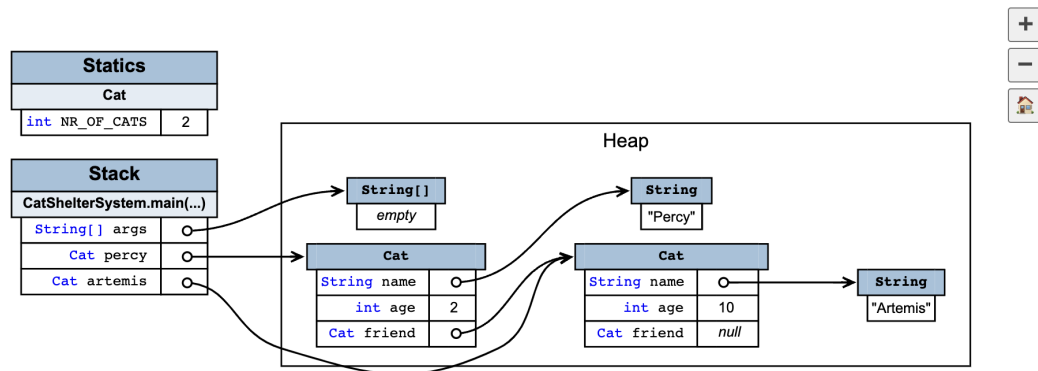


Figure 11: Memory View.

### 6.1 Detecting Hover

In order to incorporate the new feature, the existing implementation was modified. The visualization consists of rectangles that hold information about methods, local variables, static fields, and objects and their fields.

Listing 6 introduces a simplified version of how event listeners are attached to a visual element. Every rectangle represents a node that wraps an HTML hyperlink element, namely the `<a>` tag. The element’s `href` attribute contains an identifier that uniquely identifies a heap object with the prefix "o", a local variable with the prefix "l\_", a static field with the prefix "s\_" or a `this` reference with the prefix "localvar\_". In case the identifier does not have any of the aforementioned prefixes, a method is hovered.

```
d3.selectAll('g[id^="a_node"]')
  .selectAll('a')
  .on('mouseenter', function (event) {
    let hoverInfos: HoverInfo[] = []
    const id = d3.select(this).attr('href')

    // heap object
    if (id starts with 'o') {
      hoverInfos = getHeapObjectInfo(id)
```

```

    }
    // local variable, static field, or 'this' reference
    else if (
        id starts with 'l_' ||
        id starts with 's_' ||
        id starts with 'localvar_'
    ) {
        hoverInfos = getVarOrFieldInfo(id)
    }
    // method
    else {
        hoverInfos = getMethodInfo(id)
    }
    HoverSynchronizer.hover(hoverInfos)
})
.on('mouseleave', () => clearHoverHighlighting())

```

Listing 6: Hover detection in the Memory View

Depending on the hovered node, a different method is called to retrieve the necessary information passed to the `HoverSynchronizer`. The method `getHeapObjectInfo` contains various statements to fetch and summarize the hovered information about an object. Three different types of information can be accessed:

1. **Object Field:** In case the identifier contains the name of a field a `HoverInfo` of type `HoverField` is created. If the field references another object, an additional `HoverInfo` is added to the array that has the type `HoverHeapObject`.
2. **Array Cell:** When the hovered node represents the cell of an array there is one more detail besides the object ID present in the identifier string, namely the index. The retrieved information then has the type `HoverArrayCell`.
3. **Heap Object:** If nothing from point 1 and 2 applies, a `HoverHeapObject` is passed as the hovered information.

`getVarOrFieldInfo` is responsible for correctly retrieving information from hovered nodes representing local variables, static fields, or `this` references. It is similar to how data is fetched for hovered objects on the heap. However, here the approach is much simpler. Based on the prefix of the identifier, the according `HoverInfo` type is determined, which is primarily `HoverLocal`, `HoverStatic` or `HoverThisVar`. The method additionally checks if a reference to an object is present and additionally adds a `HoverInfo` of type `HoverHeapObject` to the hovered information.

The method `getMethodInfo` is pretty straightforward and creates a `HoverInfo` object with the type `HoverMethod`. The identifier of the hovered node solely contains the name of the method with which necessary properties are gathered.

## 6.2 Reacting to HoverSynchronizer

Whenever the Memory View is notified by the `HoverSynchronizer` that something is hovered, the background color of graph nodes has to be determined. For that reason, the kind of each node in the graph is identified, which is either `HeapVizVar`, `HeapVizHeapArrayElementVar`, `HeapVizHeapString` or `HeapVizHeapArray`.

- `HeapVizVar`: If the item to be highlighted has the type `HeapVizVar`, it is either a local variable, static field, `this` reference, field of a class instance or the class instance itself.
- `HeapVizHeapArrayElementVar`: If the user hovers over the cell of an array, that element is emphasized. However, if the array variable or field is hovered, every cell shall be highlighted.
- `HeapVizHeapString`: Nodes with the kind `HeapVizHeapString` are an object on the heap containing the string value of a variable or field of type `String`.
- `HeapVizHeapArray`: The same as for `HeapVizHeapArrayElementVar`, the kind `HeapVizHeapArray` represents an array object on the heap. However, in this case, the item to be highlighted is the rectangle containing the type of the array instead of the cell.

While the Tabular Statement History View primarily highlights its rows and columns with one color, this component requires two colors. The goal is to differentiate between objects that are referenced and variables or fields that reference an object.

To maintain a consistent color scheme, a yellow background is applied to hovered local variables, static fields, and object fields as well as methods, as in Figure 12. The same color is used to highlight specific table cells in the Tabular Statement History View.

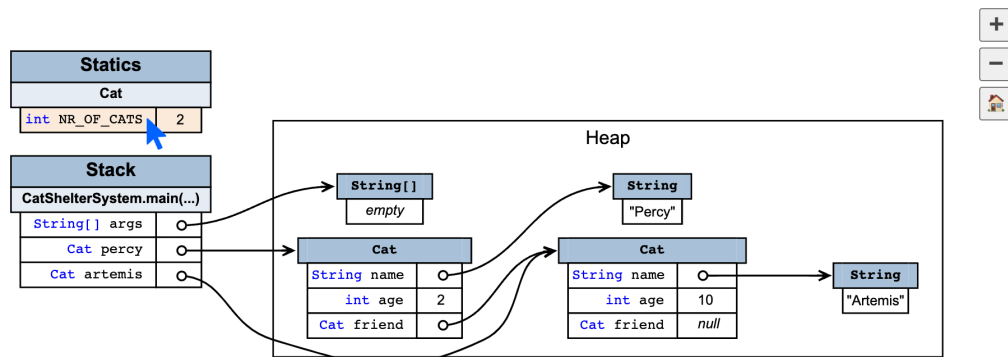


Figure 12: TheHeapVisualization: Highlighting when a static field is hovered.

If the user moves their cursor over a node that either represents an object or that references one, all nodes making out that object are colored in green as seen in Figure 13. This clarifies the relation of "referencing" and "being referenced".



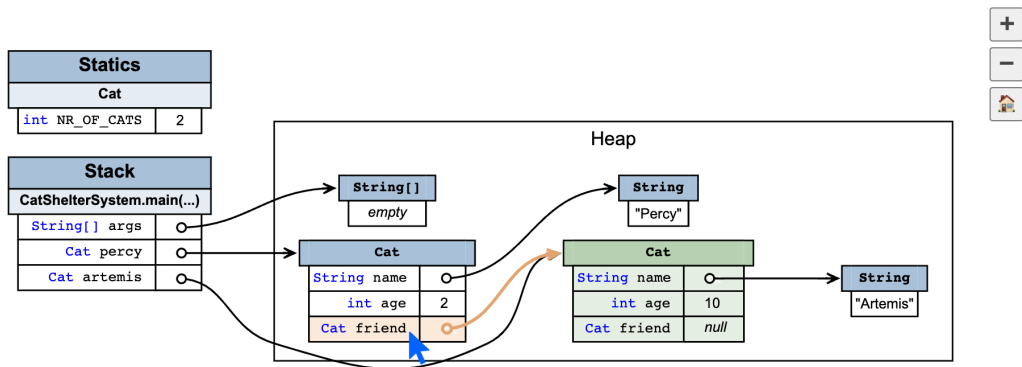


Figure 13: TheHeapVisualization: Highlighting when a field referencing an object is hovered.

The purpose of the color scheme becomes clearer in Figure 14. Assuming the user is interested in knowing by which variables or fields a certain object is referenced, solely applying the same highlighting color to the nodes might be confusing if the visualization is large. In this case, the graph renders too many nodes and the highlighting becomes unclear in terms of where the referenced object is if everything is colored the same. However, green distinctly differentiates the object from all nodes that reference it. This allows the user to identify it easily and actually focus on the interested variables and fields.

Figure 14 shows an example of how such a use case looks like. The object is explicitly emphasized in green and by following the highlighted pointers clear paths from one construct to another are determined.

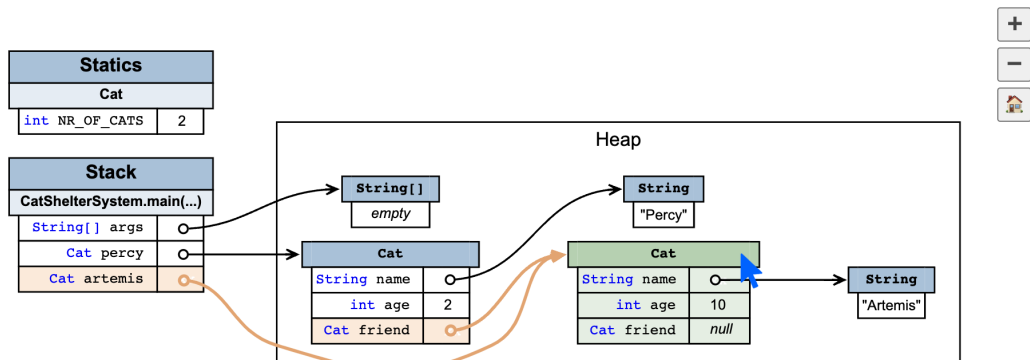


Figure 14: TheHeapVisualization: Highlighting when a heap object referenced by multiple variables and fields hovered.

### 6.3 Problems and Solutions

#### Highlighting of Arrays and the Referenced Object

When hovering a header in the Tabular Statement History View that represents an array, that

variable or field does not contain a reference to the corresponding object on the heap. Therefore, this has to be done in the Memory View by iterating over all local variables or static fields and additionally adding a `HoverInfo` of type `HoverHeapObject` to the nodes to be highlighted. As a result, the highlighting is the same as it is for all other variables or fields referencing an object, see Figure 15.

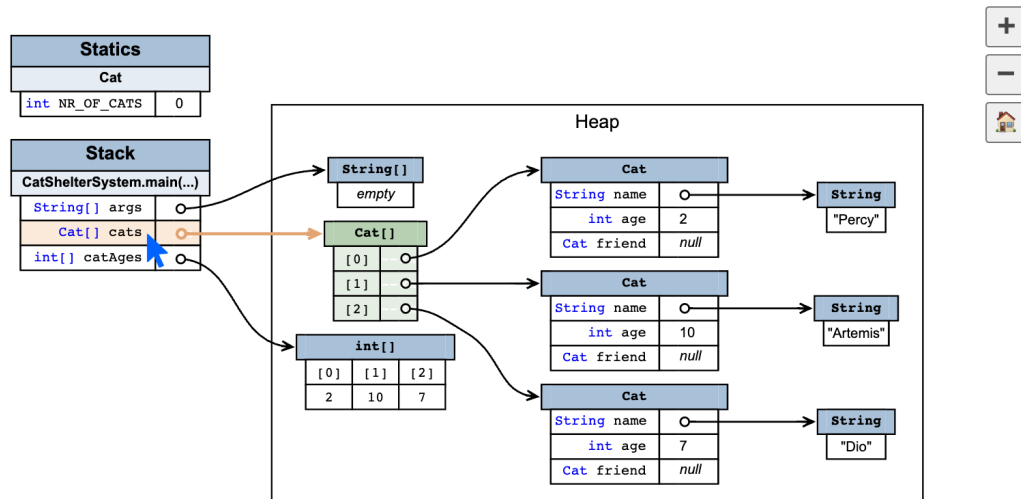


Figure 15: TheHeapVisualization: Highlighting when an array is hovered.

### Transition Error due to Usage of Wrong Events

During the beginning stages of the `HoverSynchronizer` implementation, the Tabular Statement History View and the graphs for the Linked List View or Binary Tree View did not lead to any performance issues. The Tabular Statement History View creates a HTML `<table>` that wraps the data in different rows and cells. In contrast to that the other two visualizations draw graphs using `d3-graphviz`. Nonetheless, those graphs solely animate the the program's execution steps, particularly the drawing from one step to the next animates a harmonious transition. For instance, if the `next` field of a list node is hovered in the visualization tool, concerning elements receive a different color. This initiates a re-render as well, but it is not animated. Thus no performance issues were detected.

Although the visualizations did not show anything out of the ordinary regarding user experience and efficiency, the concern about potential issues with the transitions persisted.

This concern proved to be valid when implementing the feature in the Memory View. In contrast to the other two visualizations that use `d3-graphviz`, the illustrations in this component always animate the transition. When the mouse hovers over an element, also the slightest movement triggers a re-render before the implementation of the `HoverSynchronizer`. This was due to the event listeners utilized.

Since the Tabular Statement History View used the JavaScript `mouseover` and `mouseout` events without any issues, it was assumed that these were reliable events. After incorporating the functionalities of the `HoverSynchronizer` and adding the logic for processing graph elements for the

memory, the events turned out to be quite troublesome. The two events bubble up through the entire DOM (Document Object Model) hierarchy, meaning when a child element is hovered, the event is triggered on parent elements as well [5][4]. The DOM represents an HTML document as a tree of nodes. This allows developers to access and manipulate the content of a web page. Although it is not noticeable to the naked eye, multiple re-renders are executed which throttle the performance. Alternatively, to these events, the JavaScript `mouseenter` and `mouseleave` events are solely triggered on the elements that are hovered by the user [2][3]. This reduces the number of re-renders while ensuring enhanced performance. Consequently, the previously incorporated events were replaced by the latter two.

### **Transition Error due to Number of Animated Re-renders**

Even after the modification of event listeners issues remained that influenced the performance of the educational visualization tool. This is mostly because some visualizations in JavaWiz animate transitions between re-renders. When the next render process is initiated before the current one is finished, the tool throws an error because the transition was interrupted and replaced. This could occur when the user clicks the step options of the debugger too quickly. But also hovering over different visual elements leads to error messages because the application tries to redraw components multiple times within a short timeframe. The solution was then to add a few code lines that debounce the rendering process. Debouncing is commonly used to ensure that a function, in this case, the redrawing process, is not called too frequently.

This part of the feature implementation checks if a redraw has already been initiated. If not, the visualization is animated. However, if a render is currently in process and JavaWiz tries to initiate another redraw, this new redraw, and any further redraws are delayed until the first redraw is finished. This approach reduces transition errors but cannot prevent them entirely.

## 7 Hover Synchronization in the Linked List and Binary Tree View

Equal to the Memory View, the graphs in the Linked List and Binary Tree View are drawn with d3-graphviz.

Each rectangle in Figure 16 represents a node. The fields of a node are represented by smaller rectangles within it. In a list node, typical fields are "value" and "next" which contain the reference to the subsequent node. A tree node usually has a "value" field and two additional fields, namely "left" and "right", which reference other nodes as well.

Besides that, pointers are directed from one node to another, which means they contain information about the referencing and the referenced element in the form of a "from-to" relation.

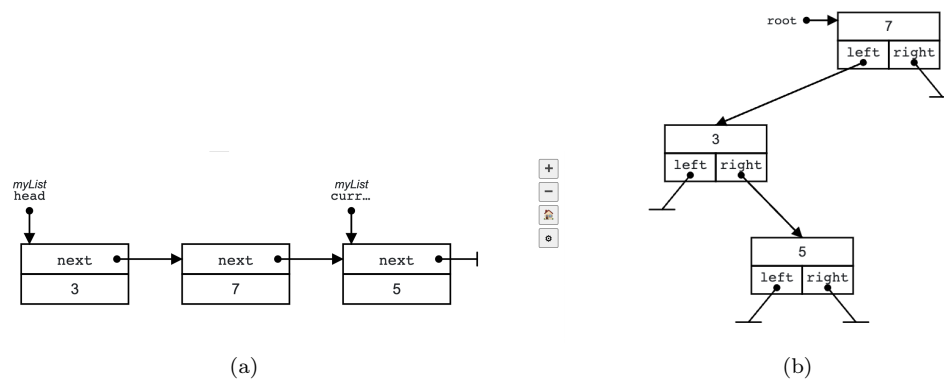


Figure 16: Linked List View and (Binary) Tree View.

If the number of connected nodes becomes too large, it might become unclear which node follows after another. Therefore, the synchronized hover highlighting feature shall provide a better overview by highlighting a certain path. This section covers the implementation of the feature in the Linked List View and Binary Tree View.

### 7.1 Detecting Hover

Section 6.1 describes how the new highlighting functionality is incorporated in the Memory View. Since both components that are described in this section are rendered with the same library, the approach is similar. An event listener is attached to a visual element that contains important information that should be fetched when hovering it.

The Linked List View listens to the hover over the "value" and the "next" field as well as pointers. If a field is hovered, the created `HoverInfo` has the type `HoverField`. Additionally, the reference of "next" is forwarded as a `HoverHeapObject` if existent.

However, when looking at pointers, they have to be distinguished into three types.

- **"next" pointer:** If the pointer between the "next" field and the node that the arrow points

to is hovered, the `HoverSynchronizer` receives two `HoverInfos`. The first is a `HoverField` representing "next" and the second information is a `HoverHeapObject` for the referenced node.

- **Field pointer:** If the retrieved data contains a `pointerId`, the hovered pointer symbolizes the field of a class. Therefore, the `HoverSynchronizer` receives a `HoverField`, which is the pointer, and a `HoverHeapObject`, which is the referenced node, as the hovered information.
- **Variable pointer:** In the event that a pointer does not have a `pointerId`, the arrow illustrates a local variable. In this case, a `HoverLocal` object is passed to the hover method instead of a `HoverField` object.

The difference between the latter two is further elaborated in Listing 7 based on a pseudo code.

```
pointerGroup.append('g')
.on('mouseenter', (_event, pointer) => {
  const hoverInfos = getListNodeInfo(pointer.referencedNode)

  // check if a pointer is hovered
  if (pointer ID) {
    hoverInfos.push(create a HoverField)
  } else {
    // otherwise a local variable is hovered
    const locals = get locals from heap

    for each local variable {
      if (unique attributes of local var === attributes of pointer) {
        hoverInfos.push(create a HoverLocal)
        break
      }
    }
  }
  HoverSynchronizer.hover(hoverInfos)
})
.on('mouseleave', () => HoverSynchronizer.clear())
```

Listing 7: Hover detection with pointers in the Linked List and Binary Tree View

The implementation of the synchronized hover highlighting feature in the Binary Tree View is almost the same as in the Linked List View. Solely the "next" field is different. As mentioned, each node in the tree graph contains a "left" and a "right" field instead. Nonetheless, the way how information is retrieved is identical.

## 7.2 Reacting to `HoverSynchronizer`

The determination of which nodes or pointers should be highlighted and what color they receive is pretty straightforward. A visual element is passed as a parameter to a method that iterates

over the items to be highlighted. In case the graph element corresponds to the hovered information, it receives a background color. The color scheme is yellow for fields or pointers that reference another node and green for the referenced node. This ensures the overall consistency of the meaning behind those colors which is explained in Section 6.2.

If the user hovers over the value field, the rectangle representing it receives a yellow background as in Figure 17.

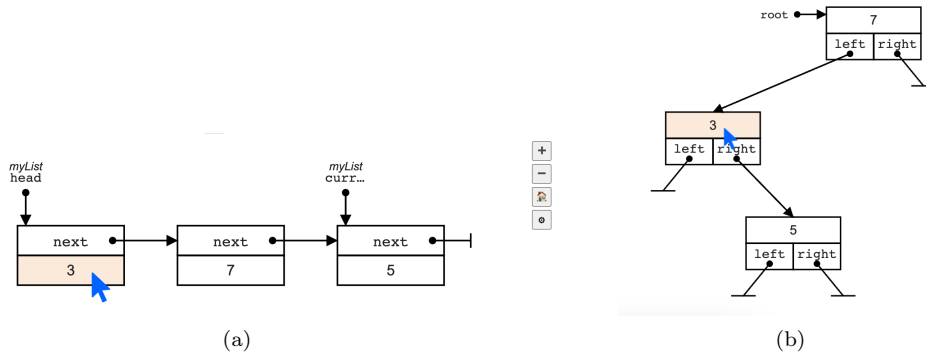


Figure 17: List View & Tree View: Highlighting when a "value" field is hovered.

When the `next` field in the linked list, or the `left` or `right` field in the binary tree is hovered, the color is the same as for the `value`. Additionally, the pointer is emphasized in yellow and the referenced node is colored in green. Figure 18 shows an example of the mentioned use case. This enables a better understanding of the relationship between two nodes and highlights the path connecting them.

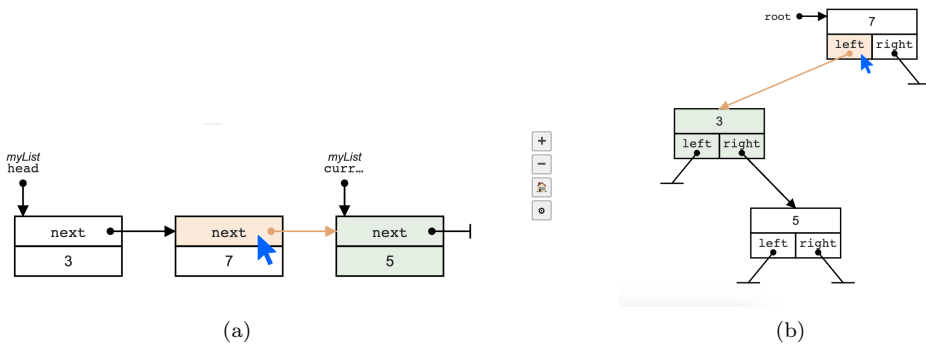


Figure 18: List View & Tree View: Highlighting when a field referencing another node is hovered.

Figure 19 resembles the highlighting of graph elements when a pointer is hovered that symbolizes a local variable or field of a class instance. Unlike Figure 18, Figure 19 shows the highlighting of elements that do not correspond to fields of a list or tree node but rather the fields of a Java

class, that, for instance, represent the list or tree themselves, or other variables.

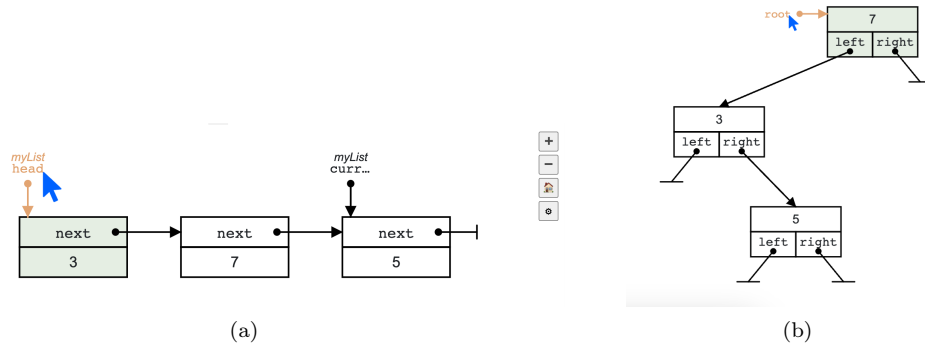


Figure 19: List View & Tree View: Highlighting when a pointer is hovered.

## 8 Usage

Previous sections covered the thought process during development and explained the concrete implementation of the hover highlighting feature. Now that a foundation has been created that helps in understanding the feature, this section covers a possible usage of JavaWiz with emphasized elements. The following code was created to demonstrate the functionalities of the new feature:

```
class Cat {
    public static int NR_OF_CATS = 0;

    private String name;
    private Cat friend;

    public Cat(String name) {
        this.name = name;
    }

    public void setFriend(Cat friend) {
        this.friend = friend;
    }
}

public class CatShelterSystem {
    public static void main(String[] args) {
        System.out.println("Welcome to the cat shelter!");

        Cat percy = new Cat("Percy");
        Cat.NR_OF_CATS++;

        Cat artemis = new Cat("Artemis");
        Cat.NR_OF_CATS++;

        Cat dio = new Cat("Dio");
        Cat.NR_OF_CATS++;

        percy.setFriend(artemis);
        artemis.setFriend(dio);
    }
}
```

Listing 8: Example of JavaWiz with the hover synchronization feature

Listing 8 shows a program containing numerous abstract ideas of Java. In particular, the file contains two classes, namely `Cat` and `CatShelterSystem`, that represent a simplified version of real-world elements. The class `Cat` portrays the animal with a name and a friend. Moreover `NR_OF_CATS` is a static field that counts the number of cats present at the shelter.



The class `CatShelterSystem` contains the `main` method, in which three `Cat` objects are created while incrementing `NR_OF_CATS`. Additionally, it determines which cats are friends.

Figure 20 illustrates the JavaWiz web application after stepping over each code line with the debugger tool. The left half shows the code editor where users can upload Java files or create and edit their code. The right side renders three visualizations, namely the Memory View at the top, the desk test in the middle, and the Linked List View at the bottom.

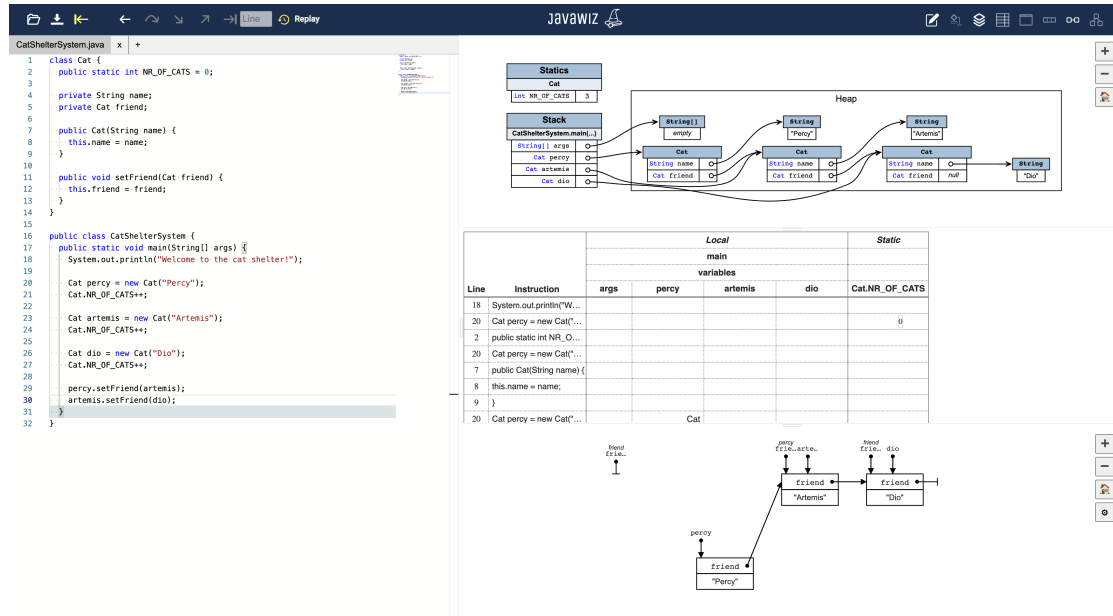


Figure 20: JavaWiz without the `HoverSynchronizer`.

JavaWiz already offers a handful of options for users to interact with elements within a visualization. It allows zooming in and out of illustrations and collapsing and expanding objects in the Memory View. However, it was not possible to directly interact with all visual elements. With the implementation of the `HoverSynchronizer` novice programmers and lecturers can emphasize certain elements, such as local variables. As a result, people are able to perceive in which components the hovered data is the same and what other relations exist, which are further explained in the subsequent sections.

## 8.1 Constructor and this Reference

As seen in Figure 21, both components display the same information considering local variables, static fields, and methods. Nonetheless, the Memory View contains a method `Constructor` of `Cat` whereas the Tabular Statement History View has a method `<init>`. Without emphasizing those two elements, experienced programmers are more likely able to tell that those two represent the same method. Nonetheless, JavaWiz is primarily targeting novice programmers. Therefore, the different naming could lead to confusion. Assuming beginners are already somewhat familiar with the term "constructor", they might still ask themselves: "Why is there an `<init>` method even though there is no such method in my code?"

With the new feature, people can hover over the constructor in the upper visualization or the initialization method in the lower component. The user interaction would then emphasize all related elements in other components as well. In this context, the constructor and `<init>` are highlighted. This provides more clarity about the identical underlying structure of those two elements.

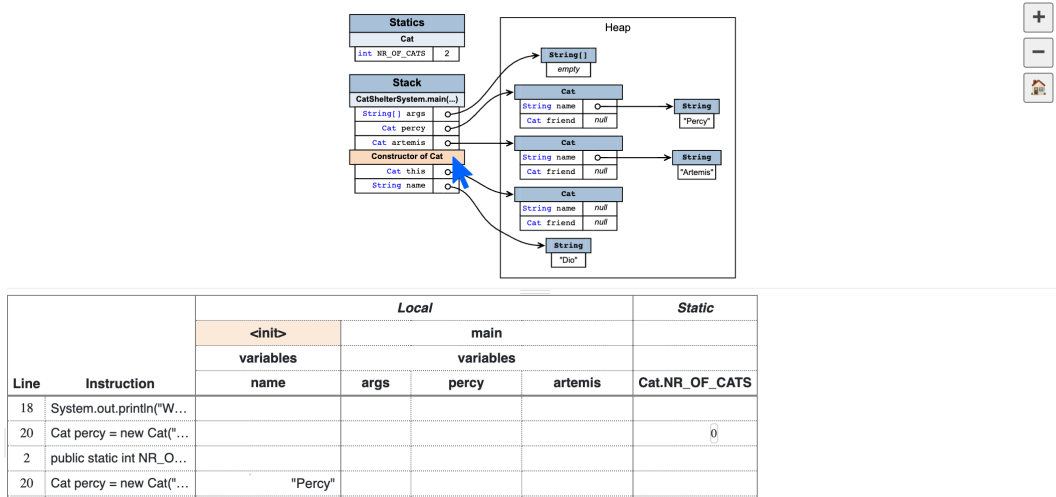


Figure 21: Highlighting when a constructor, the object initialization method, is hovered.

Whenever the user steps into a constructor while creating an object, a particular reference variable is created as well. The keyword `this` stands for exactly that. It references the current instance of a class. When calling the initialization method, the object has not been assigned to any variable or field yet. Therefore, by hovering over the reference variable, the new object that it now references by `this` is emphasized. Figure 22 essentially says that `this` points to an object on the heap. That object is responsible for setting the field `name` correctly with the value passed in the method call. After the initialization, the object is assigned to a local variable, static field, or a field of another object.

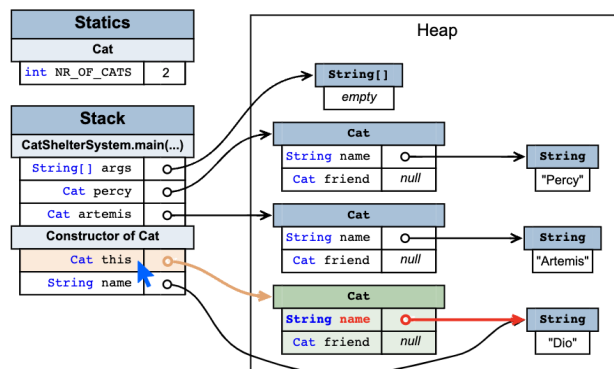


Figure 22: HoverSynchronizer: Highlighting when the `this` reference is hovered.

## 8.2 Hovering Lines and Columns in the Tabular Statement History View

Section 5 already gave a thorough explanation of how the `HoverSynchronizer` was implemented in the Tabular Statement History View. Figure 23 and Figure 24 further show how hovering different elements in the desk test affects other components.

When a user hovers over a header in the table, specifically any local variable, static field, or expression of a condition, only the according column is highlighted as seen in Figure 23. The Tabular Statement History View solely tells programmers that the variable `percy` has the type `Cat` and references an object on the heap. With the `HoverSynchronizer` remaining components that implemented it as well are notified about the user interaction. They receive the information that a local variable with the name `percy` is currently hovered. Each listener processes that information and highlights the corresponding elements in their visualization. This is especially helpful since in many cases programmers would like to know what fields the object has and which values are assigned to them. The desk test conveys the information that an object was created while the Memory View and graph in the Linked List View display the structure of that object. Users can conclude that the object represents a cat with the name "Percy" who has a friend who is also a cat.

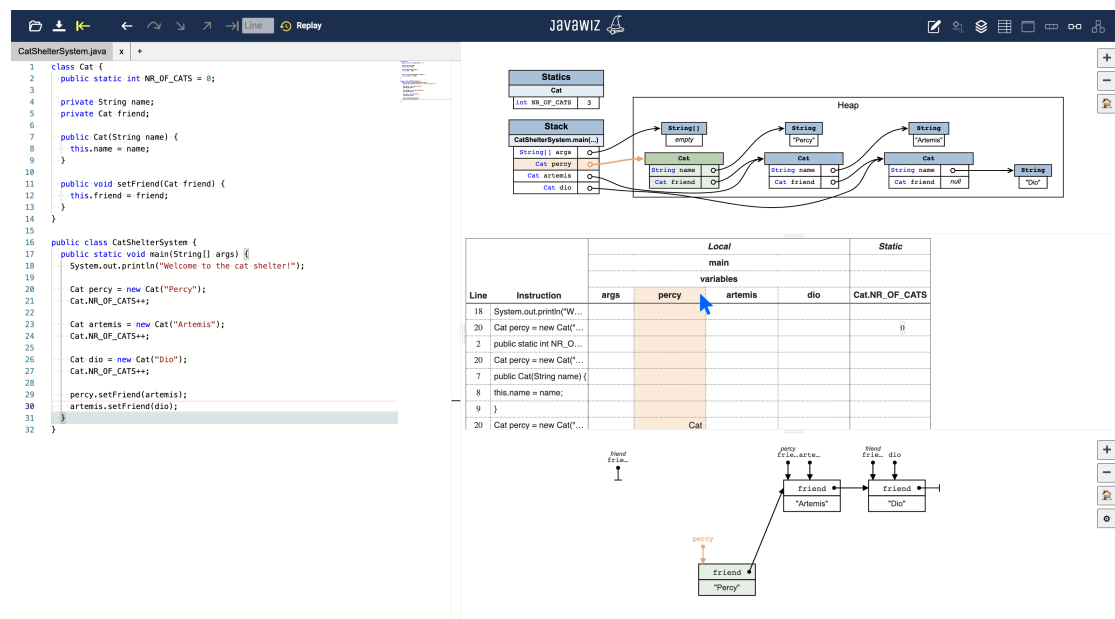


Figure 23: `HoverSynchronizer`: Highlighting when the header of a desk test column is hovered.

The visualization tool allows programmers to hover over each line in the desk test as well. This was already available before the new feature. Nevertheless, this was only visible in the said component. The hover highlighting feature now also highlights the concerning line in the code editor so people would not have to search for it but see it at first glance. Another difference introduced by the new implementation is the additional highlighting of the table column, which

Section 5 addresses. However, when comparing Figure 23 and Figure 24 one can see that there is not a big difference.

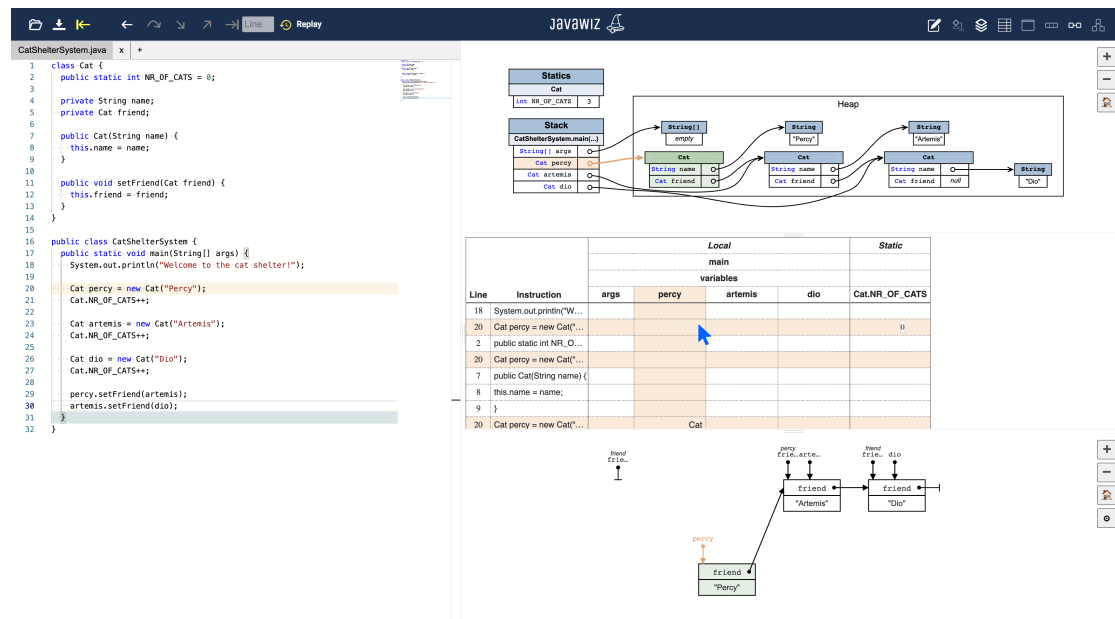


Figure 24: HoverSynchronizer: Highlighting when a desk test line is hovered.

The only noticeable difference is the additional highlighting of lines in Figure 24. Despite that the information emphasized in the Memory View and Linked List View are identical compared to before. This implies that the `HoverSynchronizer` is in fact simply responsible for notifying its listeners about the information being hovered. How that content is processed and integrated into the visualizations is different for each component.

## 9 Related Work

In this section, a few writings related to the new feature, which was implemented in the scope of this thesis, are reviewed. This review aims to provide more insight into current capabilities, highlighting key aspects and identifying limitations. Furthermore, the newly introduced JavaWiz feature is compared to existing solutions, addressing advantages and analyzing how it diverges from others.

The so-called synchronized hover highlighting feature is responsible for emphasizing the same or related elements within various visualizations of an application. In contrast to the concept of synchronized visual highlighting, the term "Visual Link" refers to the emphasis on identical or related data points that are represented differently over one or more applications [14]. Both serve the same purpose of enhancing the users' experience. Nonetheless, there are a few key differences worth mentioning.

One of the most prominent points is the opportunity for users to interact with the presented data. The **HoverSynchronizer** allows the hover over certain elements within a component which triggers the highlighting of that information in other visualizations. As a result of the hovering, graphs are adapted accordingly and guide the user's attention to the piece of information they are currently interested in.

Unlike the new JavaWiz feature, a visual link is mostly static since the alteration of the content should be avoided [14]. Related information is solely presented in such a way to guide individuals within one or more applications, such as the connection of a facility's name to their geographic location [15]. Figure 25 provides a visual example of the relations between links.

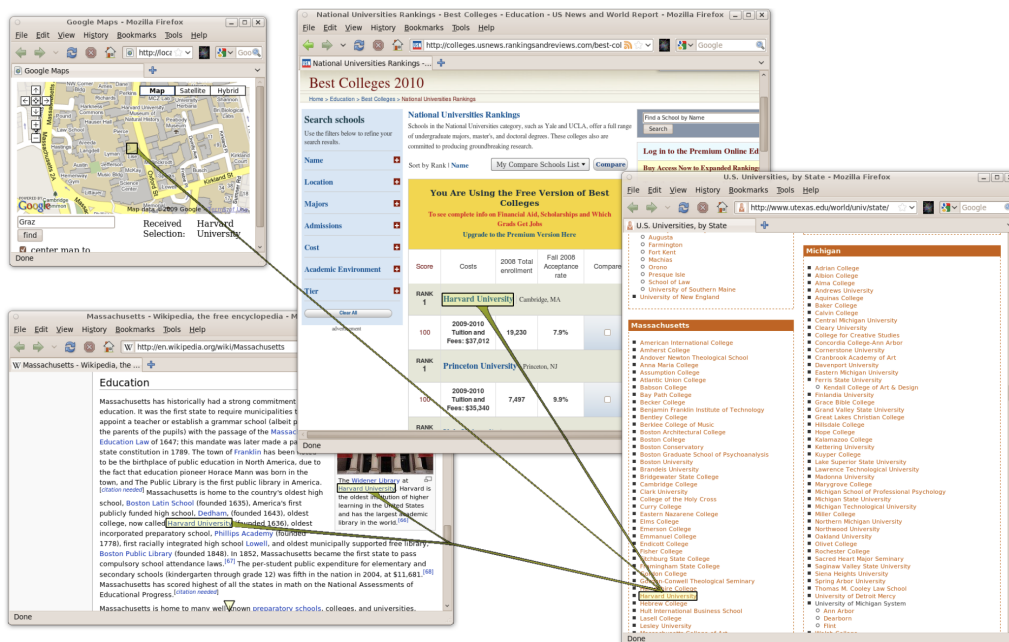


Figure 25: Representation of the relations between visual links, taken from "Visual links across applications" by M. Waldner et. al., 2010, Proceedings - Graphics Interface [15].

However, the hover highlighting feature solely determines which elements are highlighted on-demand. This is especially useful if the data becomes too large.

In conclusion, both solutions create a better understanding of the relation of certain data points. If the relation shall be showcased at all times a visual link is more efficient as it already renders the components accordingly from the beginning. However, when considering the displaying of connected information only on demand, a synchronized hover highlighting feature is the recommended approach. Contrary to the visual link that has to retrieve data and compute visual connections in every render process, those computation steps in the highlighting feature are only executed when triggered by the user through a mouse hover.

Since JavaWiz is an interactive visualization tool, the synchronized highlighting approach proves to be the better choice considering performance and efficiency.

## 10 Limitations and Future Work

This project, while successful in many respects, faced several limitations that should be acknowledged. In addition to that suggestions for future work aims at addressing those issues and improving existing solutions. Those two aspects are considered in the subsequent sections.

### 10.1 Limitations

Even though the last sections provided insight into the new hover highlighting feature, including its overall functionality and benefits, some limitations remain.

Performance is one of the primary limitations. Since JavaWiz demonstrates the control flow of a Java program, the graphs are updated with every execution step. Every step triggers a re-render process to modify the graph on the user's screen. Before the new feature was introduced, one of the few user interactions that could initiate a redraw were the step options of the debugger. Now that elements that are hovered by the user receive a different color to emphasize them, more re-renders are triggered. As a result, the performance slows down and JavaWiz throws errors if the next render process is initiated while the current one has not finished.

As mentioned in Section 6.3, performance is a valid point of concern. Although it is not ideal, the implementation of the new hover highlighting feature introduces the most promising solution as of currently. Since the visualization tool redraws the graphs after each execution step and on mouse hover a trade-off needed to be found. Therefore, by debouncing the re-rendering process in the Memory View and by choosing more reliable mouse event types to listen to the performance was enhanced. Other better-fitting solutions might exist but were not further researched due to time limitations.

### 10.2 Future Work

Keeping the performance issues in mind, a possible future work would be to improve the rendering execution for a smoother user experience. Furthermore, the implementation in the scope of this thesis includes the modification of the Tabular Statement History View, the Linked List View, the Binary Tree View, and the Memory View. The decision to solely implement the `HoverSynchronizer` in these components was made due to time limitations. A period of six months was allocated for the implementation of the feature.

Although the integration in other visualizations would have been possible in that duration, they were not chosen due to their complexity. The goal was to modify as many visualizations as possible while meeting the user's needs. The only component that required more attention was the Memory View due to the illustration of numerous different data structures.

The hover highlighting feature delivers a positive contribution to making Java programming constructs more understandable to novice programmers. Consequently, other visualizations should integrate the feature in the foreseeable future. Moreover, feedback from users will also be collected for future improvements, for instance by means of a questionnaire. Including the proposed suggestions might require the adaptation of the current logic behind the synchronization of the hover or the information structure of hovered elements. Especially the flow chart visualization could lead to obstacles due to their more complex nature. Developers have to analyze the existing source code beforehand and then figure out what information can be retrieved and how.

Another point for future improvement would be the generalization of how the hovered information is processed and sent to various visualizations. The current implementation of the new feature requires each component that implements the `HoverSynchronizer` to prepare the information of hovered elements individually. By creating common methods that enable the proposed idea, the efficiency of JavaWiz could be improved once again.



## 11 Conclusions

This thesis comprises the introduction of the new feature in JavaWiz. It allows novice programmers and lecturers to view abstract ideas and algorithms in Java programming more clearly. The practical part of the project covers the modification of the Tabular Statement History View, the Linked List View, the Binary Tree View, and the Memory View.

Although other components were not considered due to certain limitations, the ground work is present. Therefore, in future work the current implementation of the hover synchronization can be integrated into components where it has not been done yet. Slight modifications of the logic or the structure of the hovered information might be necessary for improved maintenance. More specifically, this feature was programmed with the goal of reuse and reduced complexity while enhancing the user experience and performance.

In conclusion, the hover highlighting feature provides a solution that further supports beginner programmers and other people who are interested in learning concepts of Java programming. JavaWiz now not only guides its users through how a Java program executes code but also shows how, for example, instances of classes and variables or fields relate to each other. Further refinements suggested through systematic user feedback can improve the perception of the educational visualization tool based on real-world usage patterns. By incorporating surveys and usability studies, developers can better align the tool with the needs and preferences of its users. Nonetheless, the present version of the new feature already introduces concepts that open the door for future utilization and enhancement.



## List of Tables

1	Structures of the hovered information shared in multiple visualizations. . . . .	14
---	--	----

## List of Figures

1	Example of a local variable and static field referencing the same heap object. . .	5
2	JavaWiz web application. . . . .	8
3	Idea of a synchronized hover in JavaWiz. . . . .	10
4	<code>HoverSynchronizer</code> class. . . . .	11
5	<code>HoverInfo</code> interface. . . . .	11
6	Tabular Statement History View. . . . .	16
7	Tabular Statement History View: Highlighting when a method is hovered. . . . .	20
8	Tabular Statement History View: Highlighting of a column when a table header is hovered. . . . .	20
9	Tabular Statement History View: Highlighting when a line is hovered. . . . .	21
10	<code>HoverSynchronizer</code> : More information is displayed when a primitive value in the Tabular Statement History View is hovered. . . . .	21
11	Memory View. . . . .	22
12	<code>TheHeapVisualization</code> : Highlighting when a static field is hovered. . . . .	24
13	<code>TheHeapVisualization</code> : Highlighting when a field referencing an object is hovered. . . . .	25
14	<code>TheHeapVisualization</code> : Highlighting when a heap object referenced by multiple variables and fields hovered. . . . .	25
15	<code>TheHeapVisualization</code> : Highlighting when an array is hovered. . . . .	26
16	Linked List View and (Binary) Tree View. . . . .	28
17	List View & Tree View: Highlighting when a "value" field is hovered. . . . .	30
18	List View & Tree View: Highlighting when a field referencing another node is hovered. . . . .	30
19	List View & Tree View: Highlighting when a pointer is hovered. . . . .	31
20	JavaWiz without the <code>HoverSynchronizer</code> . . . . .	33
21	Highlighting when a constructor, the object initialization method, is hovered. . .	34
22	<code>HoverSynchronizer</code> : Highlighting when the <code>this</code> reference is hovered. . . . .	34
23	<code>HoverSynchronizer</code> : Highlighting when the header of a desk test column is hovered. . . . .	35
24	<code>HoverSynchronizer</code> : Highlighting when a desk test line is hovered. . . . .	36
25	Representation of the relations between visual links, taken from "Visual links across applications" by M. Waldner et. al., 2010, Proceedings - Graphics Interface [15]. . . . .	37

## Listings

1	How to access a local variable, static field or heap object . . . . .	6
2	<code>HoverMethod</code> object. . . . .	17
3	<code>HoverLocal</code> object. . . . .	17
4	<code>HoverCondition</code> object. . . . .	17
5	Information received by JavaWiz when hovering a desk test line. . . . .	18

6	Hover detection in the Memory View . . . . .	22
7	Hover detection with pointers in the Linked List and Binary Tree View . . . . .	29
8	Example of JavaWiz with the hover synchronization feature . . . . .	32

## References

- [1] M. Bostock. D3 - the javascript library for bespoke data visualization, 2011. Accessed: 27.08.2024.
- [2] M. W. Docs. mouseenter event. [https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseenter\\_event](https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseenter_event), 2024. Accessed: 21.07.2024.
- [3] M. W. Docs. mouseleave event. [https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseleave\\_event](https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseleave_event), 2024. Accessed: 21.07.2024.
- [4] M. W. Docs. mouseout event. [https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseout\\_event](https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseout_event), 2024. Accessed: 21.07.2024.
- [5] M. W. Docs. mouseover event. [https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseover\\_event](https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseover_event), 2024. Accessed: 21.07.2024.
- [6] M. R. et al. Scratch: A visual programming language for children. <https://scratch.mit.edu/>, 2007. Accessed: 26.08.2024.
- [7] I. for System Software. Javawiz - visualisierungstool für programmieranfänger. <https://ssw.jku.at/Research/Projects/JavaWiz/>, 2024. Accessed: 01.07.2024.
- [8] E. K. S. N. G. W. J. Ellson, E. Gansner. Graphviz - graph visualization software, 2004. Accessed: 27.08.2024.
- [9] M. Jacobsson. d3-graphviz. <https://github.com/magjac/d3-graphviz>, 2018. Accessed: 14.07.2024.
- [10] A. Kern. Javawiz – a visualization tool for software development education. Master’s thesis, Johannes Kepler Universität Linz, Linz, Austria, Aug. 2023.
- [11] M. Kölling. Greenfoot: A java interactive development environment for education. <https://www.greenfoot.org/>, 2006. Accessed: 26.08.2024.
- [12] R. T. M. Corney, D. Teague. Engaging students in programming. In *Processings of the Twelfth Australasian Computing Education*, pages 63–72, Brisbane, Australia, 2010. Australian Computer Society, Inc.
- [13] J. R. M. Kölling. Bluej: A free java development environment designed for beginners. <https://www.bluej.org/>, 1999. Accessed: 26.08.2024.
- [14] M. S. A. L. A. D. S. M. Steinberger, M. Waldner. Context-preserving visual linkss. *IEEE transactions on visualization and computer graphics*, 17:2249–2258, 2011.
- [15] A. L. M. S. D. S. M. Waldner, W. Puff. Visual links across applications. In *Proceedings - Graphics Interface*, NordiCHI, pages 129–136, Ottawa, Ontario, Canada, 2010. ACM.
- [16] Microsoft. Monaco editor. <https://github.com/microsoft/monaco-editor>, 2015. Accessed: 14.07.2024.

- [17] E. M. R. P. A. Ertmer, S. Gopalakrishnan. Technology-using teachers: Comparing perceptions of exemplary technology use to best practice. *Journal of Research on Computing in Education*, 33(5):19–25, 2001.
- [18] J. M. Spector. The changing nature of educational technology programs. *Educational Technology*, 55(2):19–25, 2015.

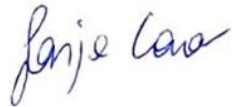


## Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Linz, Date

A handwritten signature in blue ink, appearing to read "Sanje Gao". The signature is written in a cursive style with a long, sweeping underline.