B.Comp. Dissertation

Explicit-Control Evaluator for Java

By

Liew Xin Yi

Department of Computer Science

School of Computing

National University of Singapore

2023/2024

Project No: H0411170
Advisor: Assoc Prof Martin Henz

Abstract

Source Academy is a web-based computer-mediated environment for learning programming. It is used at NUS and Uppsala University and by readers of the online edition of the textbook Structure and Interpretation of Computer Programs, JavaScript Edition. Source Academy supports sublanguages of JavaScript, TypeScript, and Scheme using an explicit-control evaluator and concomitant environment model visualizer that graphically represents the data structures evolving in a running program.

This project develops a web-based explicit-control evaluator for a suitably chosen sublanguage of Java and integrate it in Source Academy. After parsing a given Java program, the evaluator will represent the state of the running system using an control, an environment, a stash, and a class store. The execution of the program is then seen as a sequence of states. The project will visualize this state in the style of the existing environment visualizer of Source Academy.

The result of the project will be an environment for running Java programs suitable for first-year computer science students. The mental model encouraged by the visualizer avoids knowledge of the Java Virtual Machine, which provides an obstacle for many learners.

Subject Descriptors:

Languages
Design
Implementation

Keywords:

Programming Languages & Systems, Web-based Applications,
Data Structure and Algorithms, eLearning

Acknowledge

I would like to express my heartfelt gratitude to several individuals for supporting me through the course of this project.

First and foremost, I wish to thank my project supervisor, Prof. Martin Henz, for guiding me throughout this project. His knowledge and wisdom in the programming language field have unexpectedly piqued my interest in the field during my last year of CS education. His aspiration and enthusiasm of making a difference in education and learning has inspired me to think the same when developing the high-level conceptual ideas in this project.

Next, I would like to thank Prof. Prateek Saxena, the main evaluator of this project, for the feedback that he gave during the continuous assessment.

I also wish to thank Lai Mei Tin and Bryan Loh for the time and effort in discussions like AST structures, integration as well as occasional reviews and debugging. Their presence has provided some form of companionship to this individual project.

Finally, I would like to thank the Source Academy team for their support when integrating my code into the Source Academy codebase.

Table of Contents

1 Introduction

1.1 Educational Motivation

Java, as an object-oriented and statically typed programming language, introduces various concepts and posses a certain level of complexity. Educators wield the responsibility to provide the appropriate level of abstraction when it comes to teaching, particularly first year students. It is undesirable to bombard students with unnecessary details which blurs the emphasis on core concepts, what more resulting in an inaccurate mental model where more time and effort are invested to unlearn wrong knowledge. The unnecessary details for the case of Java are the Java compiler and the Java Virtual Machine.

Programming dynamics being a threshold knowledge (Sorva, 2013), is one of the most difficult topics for students. As OOP models real-world objects, while students and educators can leverage such correspondence to understand how program executes, intuition often fails when it comes to more advanced concepts, or sometimes possibly simple but uncanny programs. Pedagogical tools then come into place to assist in solidifying concepts taught in class. This dissertation introduces an explicit-control evaluator as a teaching tool for the Java programming language.

1.2 Notional Machine

Notional machine is the academic term of the CSE/CSEC machine which will be introduced later. The purpose of a notional machine is to explain program execution. While many may associate notional machine as mental model in layman term, there is a distinction between the machine itself and its visualization counterpart. This project proposes both a machine and its corresponding visualization.

While there is an existing academic debate on the choice of adoption between procedural languages and object-oriented languages in CS1 courses (Sorva, 2013), a traditional procedural-first approach is favoured. This is because it is believed that object-oriented constructs extend from procedural constructs. This is also the case in NUS, an imperative language, i.e., Source – a JavaScript sublanguage, is used in their CS1 course (CS1101S Programming Methodology, n.d.), and the CS2 course (CS2030S Programming Methodology II, n.d.), shifts to an object-oriented language, i.e., Java.

Sorva suggests that an object-oriented notional machine effectively requires at least two different notional machines in order to explain program execution clearly, especially in object-oriented CS1 courses. The two notional machines are an extended version of an imperative

notional machine and a notional machine of higher level of abstraction describe object interaction. Such an claim is exemplified in various online Java tutorials where control flow constructs, e.g., variables, conditionals, loops, etc., are discussed before classes and objects are introduced (W3Schools, n.d.) (GeeksforGeeks, n.d.) (Programiz, n.d.).

While there are also debates around students should be free to have their own interpretation of a certain knowledge (Sorva, 2013), we believe that providing a standard guide would prevent students from deviating from the correct mental model and ended up investing more time to recover from past mistakes. This is also the motivation behind educational research. Note that this does not mean that students are prevented from forming the own mental model but only serves as a guide.

1.3 Structural Operational Semantic

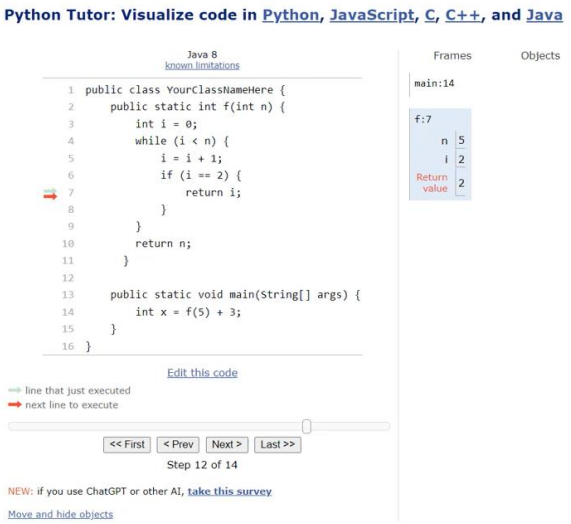
The underlying semantic framework adopted by the explicit-control evaluator is the structural operational semantics. In programming language theory, semantics provide meaning to programs, and these semantics include big-step denotational semantics and small-step virtual machine-based semantics. Structural operational semantics, on the other hand, is deemed to be at the most appropriate level of abstraction as a mixture of language construct fragments and virtual machine instructions is used. The retainment of language constructs and introduction of machine instructions serve as a bridge between syntax and semantics and allows advance constructs to be unfolded sufficiently.

2 Related Works

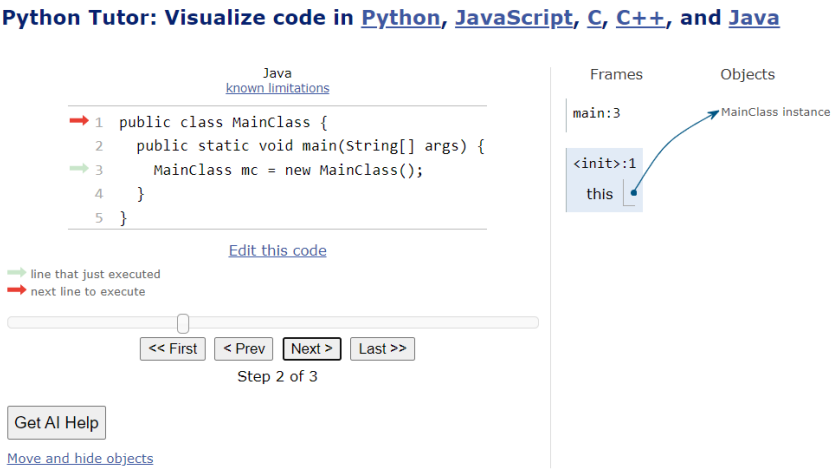
2.1 PythonTutor

PythonTutor is the only existing web-based Java visualizer. However, this solution has been heavily criticized for its approach in visualizing an imperative notional machine for the case of Python, e.g., lack of explicit control on conditional return. A stack-based discipline is also insufficient to explain more advanced concepts such as closures. An environment is required instead. In fact, a stack-based discipline is considered harmful when depicting execution of imperative program execution (John Clements, 2022).

Less effort has been made for Java as an object-oriented language, e.g., the frames and the environment are lumped together, low-level JVM specific class and instance initialization methods, i.e., <*clinit*> and <*init*> respectively, are reflected, missing inherited fields of the same name in objects, etc.



Lack of explicit control on conditional return.



JVM <init> initialization method exposed.

2.3 Novis in BlueJ

While Novis provides a decent high level notional machine to visualize object interaction, the granularity is too coarse to discuss object-oriented language concepts in depth. However, in order to fully exploit the features provided by a programming language, concepts must be discussed and understood sufficiently.

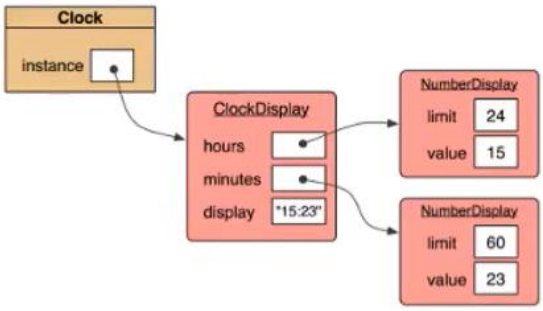


Figure 2: Representation of object references.

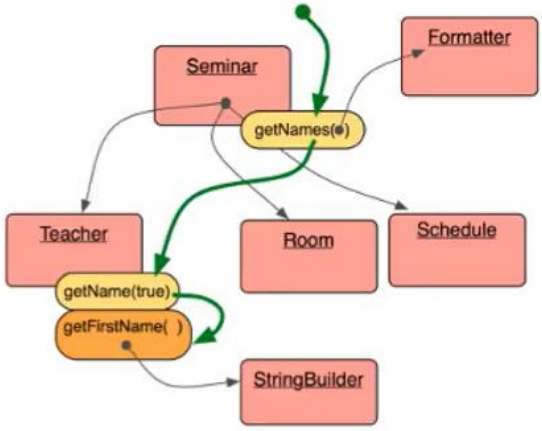


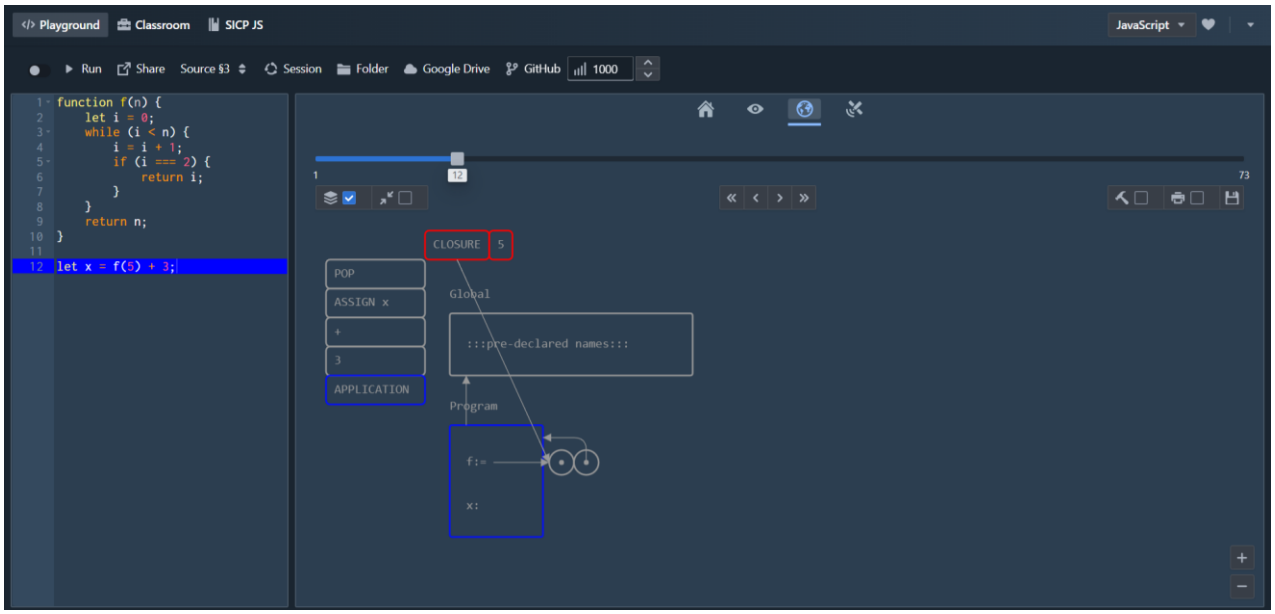
Figure 4: An active method call chain with object parameters.

2.3 Source Academy & Source §4 CSE Machine

Source Academy (SA) is an immersive online experiential environment for learning computational thinking, where users can write and run programs in web browsers (Boyd Anderson, 2023). The platform is primarily used in the first introductory programming course, a.k.a. CS1 course, i.e., CS1101S Programming Methodology, at the National University of Singapore (NUS). The course follows closely to the classic textbook, i.e., Structure and Interpretation of Computer Programs (SICP) textbook (Martin Henz, n.d.)

The Source §4 CSE Machine, as per its name, constitutes three components, i.e., control, stash, and environment. Both control and stash are stacks. The control keeps track of the execution flow by holding lines of code to be executed and intermediate instructions when destructing a particular line of code. The stash stores intermediate results when evaluating expressions. The environment, on the other hand, contains method frames and data structures.

Source §4 is a sublanguage of JavaScript that does not include object-oriented constructs. However, the Java CSEC Machine is designed to extend from the Source §4 CSE Machine, in other words, the handling of the imperative constructs is reused as much as possible.



Source §4 CSE Machine in Source Academy.

3 Java CSEC Machine

Unlike the Source §4 CSE Machine, the Java CSEC Machine constitutes three main components, i.e., control, stash, environment, and class store.

There had been multiple names mentioned thus far to refer to seemingly the same machine, e.g., Explicit Control Evaluator (ECE), Control Stash Environment Machine (CSE Machine), Control Stash Environment Class Machine (CSEC Machine). To clear any potential confusion, ECE was named in spirit of making program evaluation explicit. CSE Machine, inspired from Landin's SECD Machine, was named in spirit of making the components of the machine explicit. Finally, CSEC Machine was chosen to emphasize the need of an additional component to explain program execution of an object-oriented programming language like Java.

On first thought, classes may seem to be part of the Environment, where classes are defined in the global environment. However, environment is used to contains runtime entities, but classes are used statically and during runtime. Particularly, classes define types, and these types are used statically, e.g., method overloading and dynamically, e.g., method overriding.

3.1. Java Sublanguage Scope

The Java notional machine is aimed to be defined and implemented in incremental sublanguage scope, as per the approach adopted in the SA CSE machine.

The initial scope includes top level classes, objects, constructors, class and instance fields, class and instance methods, local variable declaration statements, assignment statements, method invocation statements, primitive integer and user-defined reference types, and only public modifiers. The exact syntactic constructs supported are listed comprehensively in the Appendix.

Although a more intuitive/logical initial scope would be the scope of the current SA CSE machine, which consists of imperative language constructs, we decided to proceed to deal with the core programming paradigm of Java – OOP. Nevertheless, we noted that adaptations are required for the existing JS notional machine, e.g., as a typed language, unfolding of assignments of values of a primitive subtype to variables of a primitive supertype would include an additional conversion step.

3.2 Design Principles

Apart from defining a sound and robust notional machine with respect to the Java Language Specification (James Gosling, 2023), here are some of the design principles that are taken into

consideration when designing the machine as well as its visualization system. The machine is not simply breaking down existing knowledge but to highlight the core concepts to students.

Explicit: As per the title of this project, we aim to make things as explicit as possible. For developer ease, syntactic sugar is provided by programming languages for ergonomical reasons, e.g., reduced development time and concise code for readability. However, under the hood, compiler injects the necessary code into user code to run the program. Such implicitness often hinders students from understanding the concept in depth and hence, we want to design against anything implicit. Approaches such as construct augmentation and preprocessing are used extensively.

Minimal: As Einstein once said, things should be explained simple but not simpler. Here an elegant solution is advocated. The concept of logic reuse and building on existing knowledge should be adopted as this bridges the gaps between existing and new knowledge which help in connecting the dots. Apart from breaking down more complex constructs into simpler constructs, another technique would be to transform one construct into another.

Evocative: As the attention span of the human mind is relatively short, visualization always helps in understanding better. The proposed visualization aims to paint a vivid picture in the mind of students when interpreting a program in their mind. Such a visualization aids students in unfolding the underlying mechanism during program execution. With sufficient repetition, the core concepts will be engrained in their mind. On a side note, this also brings up the necessity of being careful when designing such visualization as as mentioned, the time and effort to undo a wrong knowledge is usually longer than learning a new one.

While an informal structural operational semantics of the Java CSEC Machine is attached in the Appendix B, below are some highlights of the machine features that exemplifies the design principles above.

3.3 Objects

Objects are first-class citizens in object-oriented programming. Hence its importance dictates us to study them in-depth.

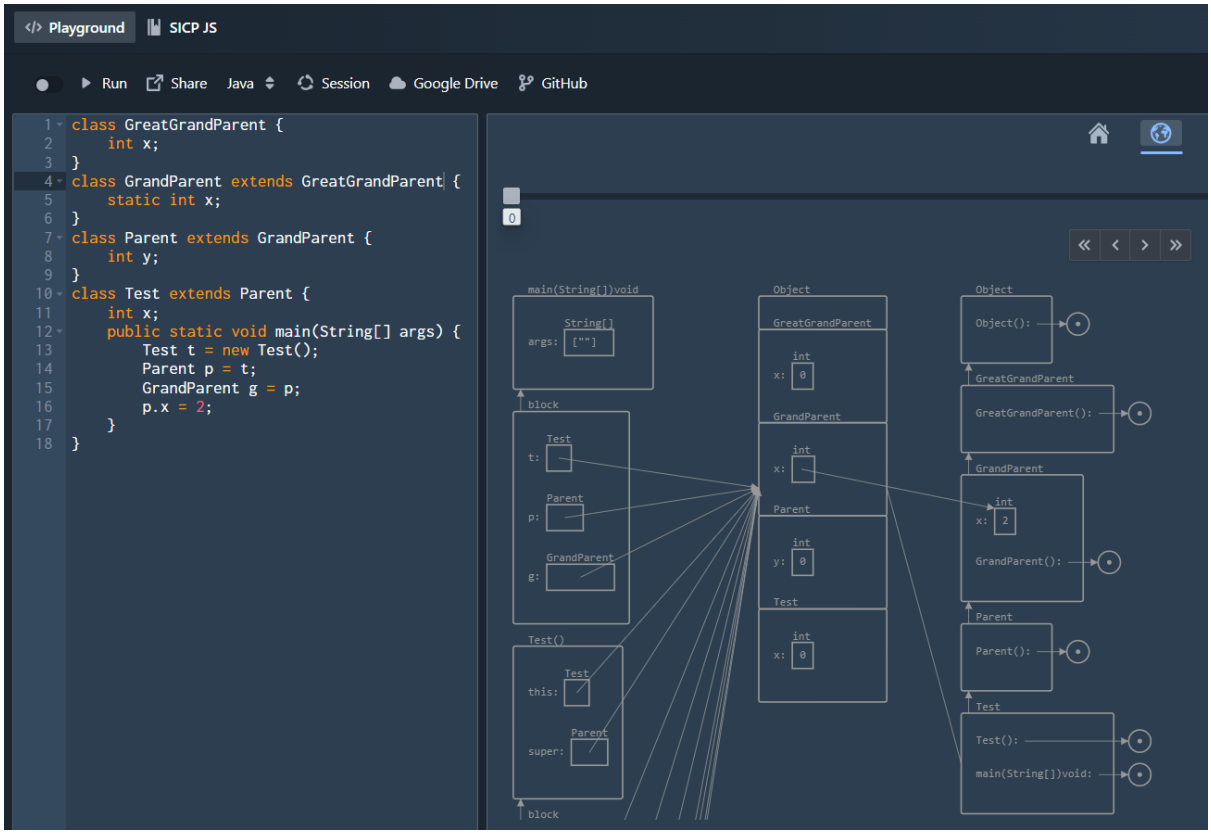
3.3.1 Object Frames

It is possible to declare instance fields with the same name as inherited fields – this is called hiding. It is also possible to access hidden fields using typing relations. This is one of the cross sections of typed and object-oriented constructs of Java.

When a new object is created, declared instance fields along with all inherited instance fields are declared and initialized. To explicitly portray the existence of inherited instance fields, objects are represented by multiple frames, where the number of frames equals 1 + the number of superclasses of the object's class.

Declared instance fields are stored in the respective frame while class fields are referenced. The presence of class field references helps in tracing name resolution.

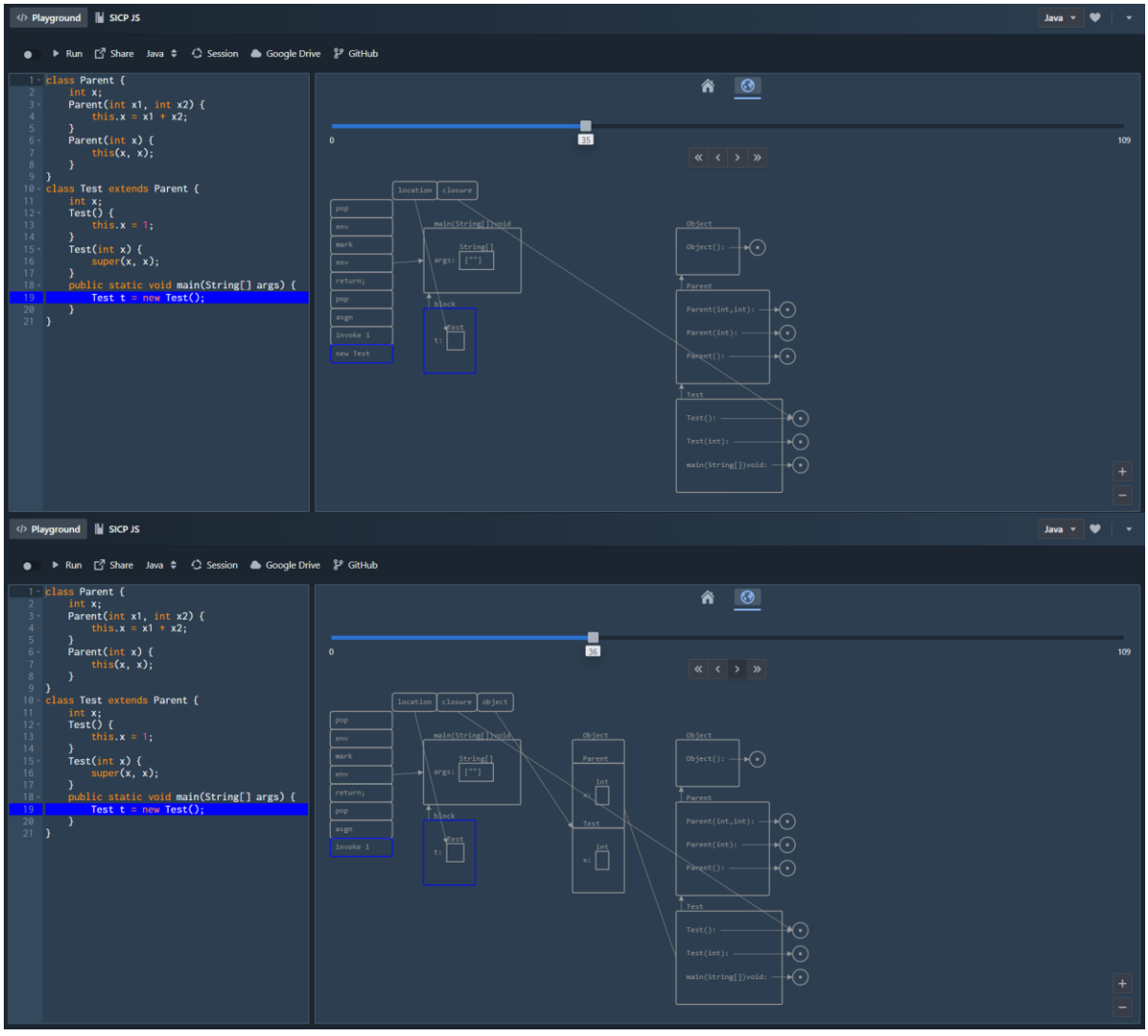
Such a representation reflects the class hierarchy within the object itself. Such a representation also avoids name clashes.



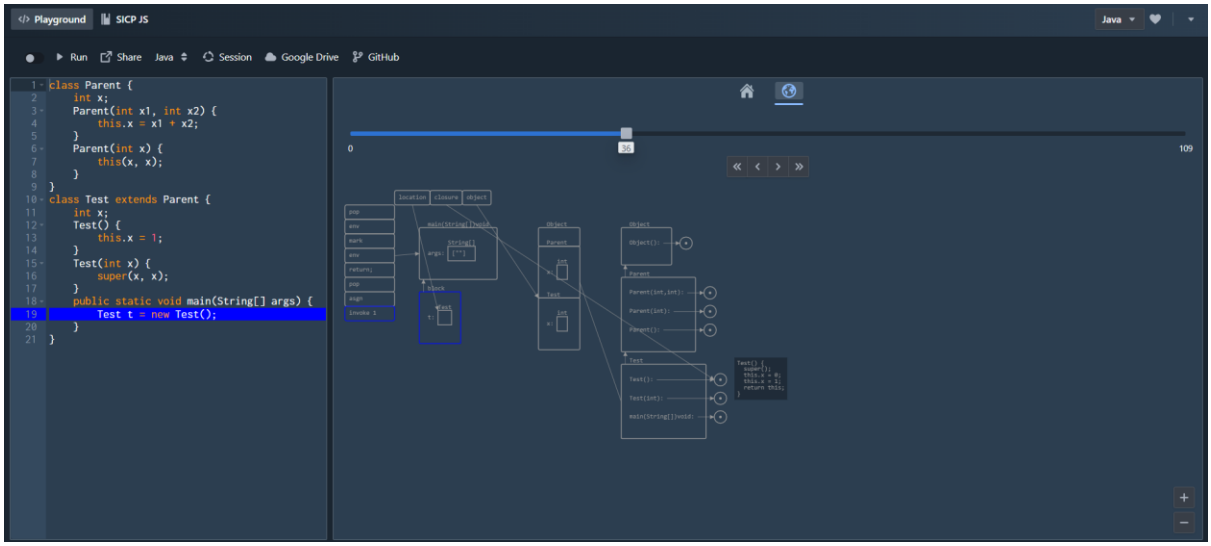
Objects represented by multiple frames.

3.3.2 Constructor not Constructor

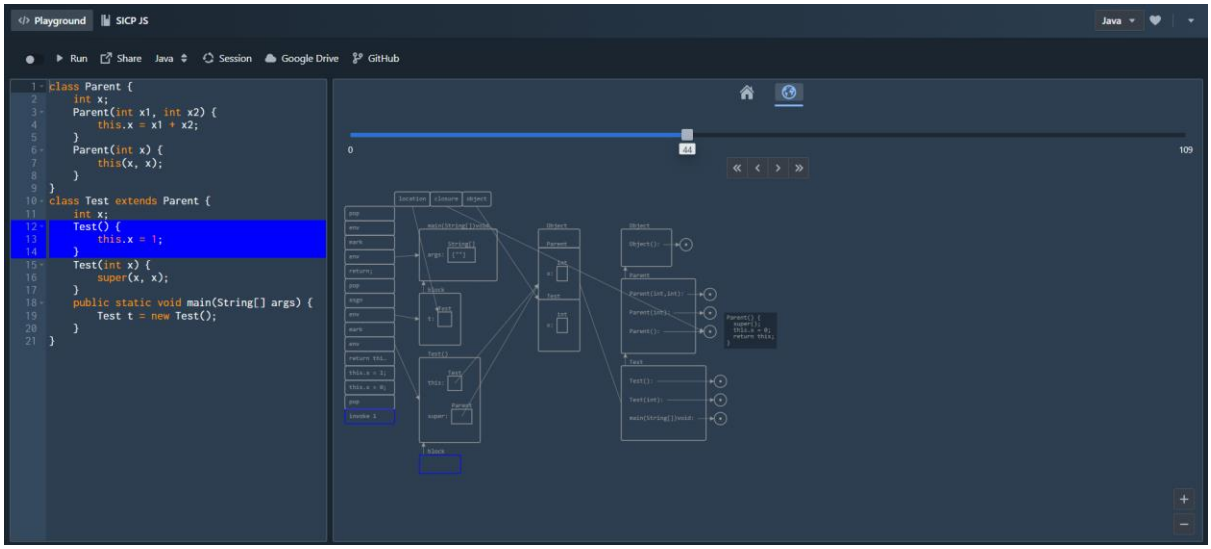
Constructor is an initializer; the new keyword is the actual constructor. The super() invocation in every constructor whose class inherits another classes seemingly implies that multiple object is being constructed. However, this is not the case – only 1 object with uninitialized instance fields is being constructed when the new keyword is being evaluated. The instance fields are then initialized when the corresponding constructor is being called.



Only 1 obejct of class Test is created.

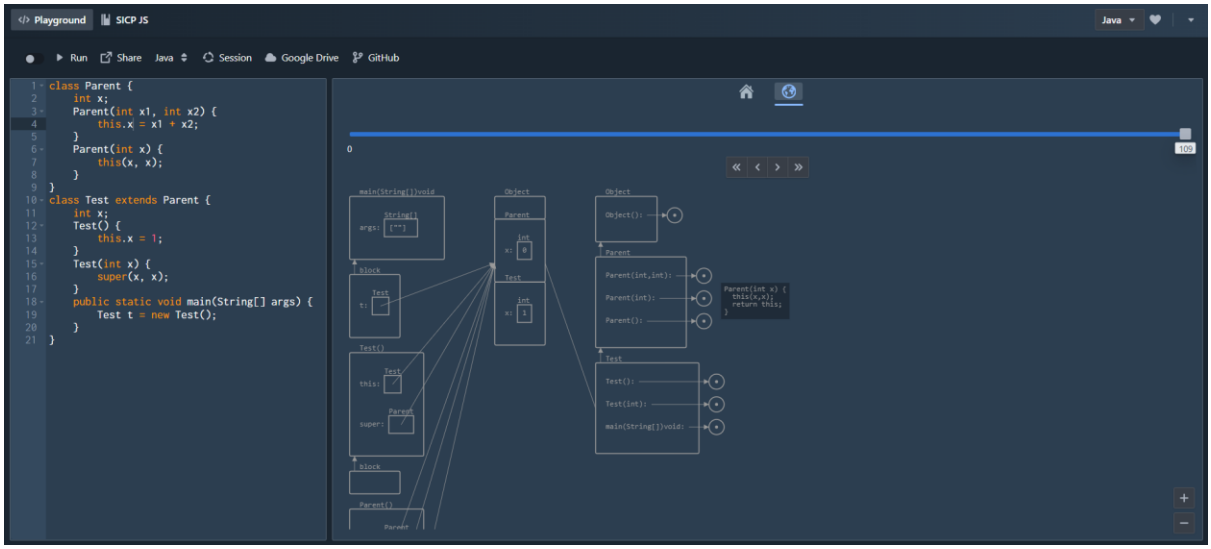


Instance field x of class Test is initialized in Test() constructor.



Instance field x of class Parent is initialized in Parent() constructor.

Also, no field initialization assignments are inserted if an alternate constructor invocation is present (James Gosling, 2023).



No field initialization assignments are inserted into constructor body as an alternate constructor invocation is present.

3.4 Name Resolution

Often, name resolution is not well-discussed during object-oriented courses. But there are interesting insights when we inspect name resolution a little more in-depth.

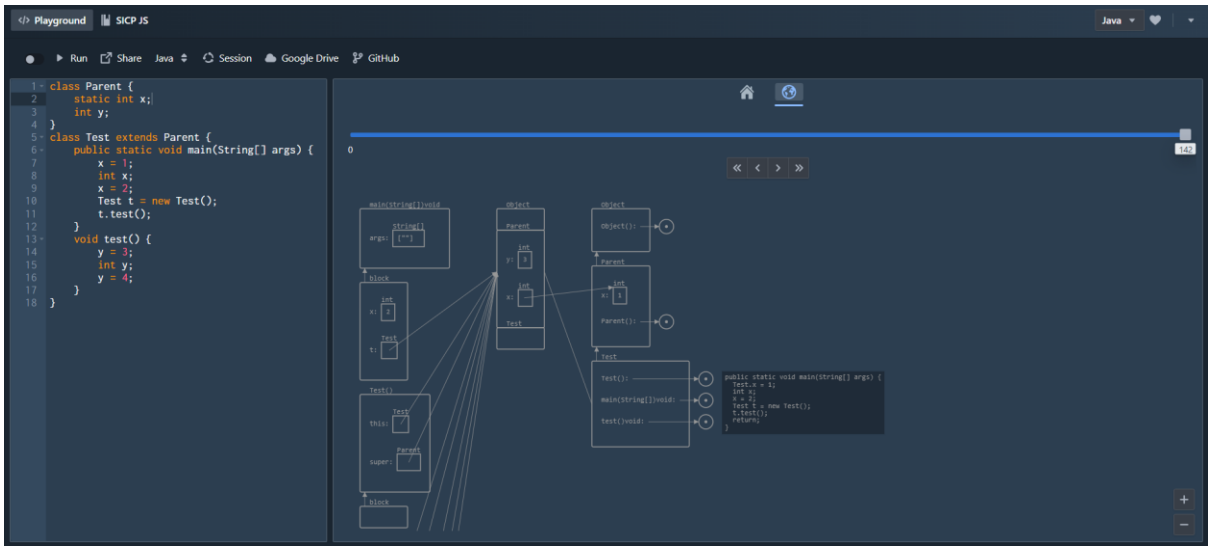
In object-oriented languages, we have classes and objects. In addition to local variables, we now have class and instance fields. Unlike local variables which can only be referred using simple names, fields can be referred using both simple names and qualified names (or dot notation). In other words, simple names can refer to either local variables or fields.

3.4.1 Separating Lexical Scoping and Field Access

Using simple names to refer to fields is syntactic sugar. As part of the desugarization process, we augment all non-local variable and non-parameter simple names with qualifiers. The qualifier prepended depends on the context, whether the name is within an instance or class method, i.e., the this keyword is prepended within instance methods while the class name is prepended within a class method. This pre-processing step is relatively straightforward as we just need to scan the block to look for any non-local variable non-parameter simple names and process accordingly. However, care should be taken to ensure the scanning is done sequentially to adhere to the scope of local variable declaration, i.e., the rest of the block starting with the declaration's own initializer and including any further declarators to the right in the local variable declaration statement (James Gosling, 2023). Interestingly, this makes the distinction between lexical scoping and field access clearer, where all simple names now refer to local variables whereas fields are referred to via qualified names only. This also bridges well with the knowledge regarding lexical scoping gained from imperative programming.

On a side note, the shadowing quirk of object-oriented programming is avoided with just a simple pre-processing.

Similarly, there are simple method names and qualified method names. For the case of method names, the pre-processing is much simpler as simple method names are purely syntactic sugar for qualified names, unlike there is semantic difference for simple names for local variables. Hence, a simple pre-processing step of prepending simple method names in instance methods with the this keyword and prepending simple method names in class methods with the class name.



Prepending simple names in class methods with class name

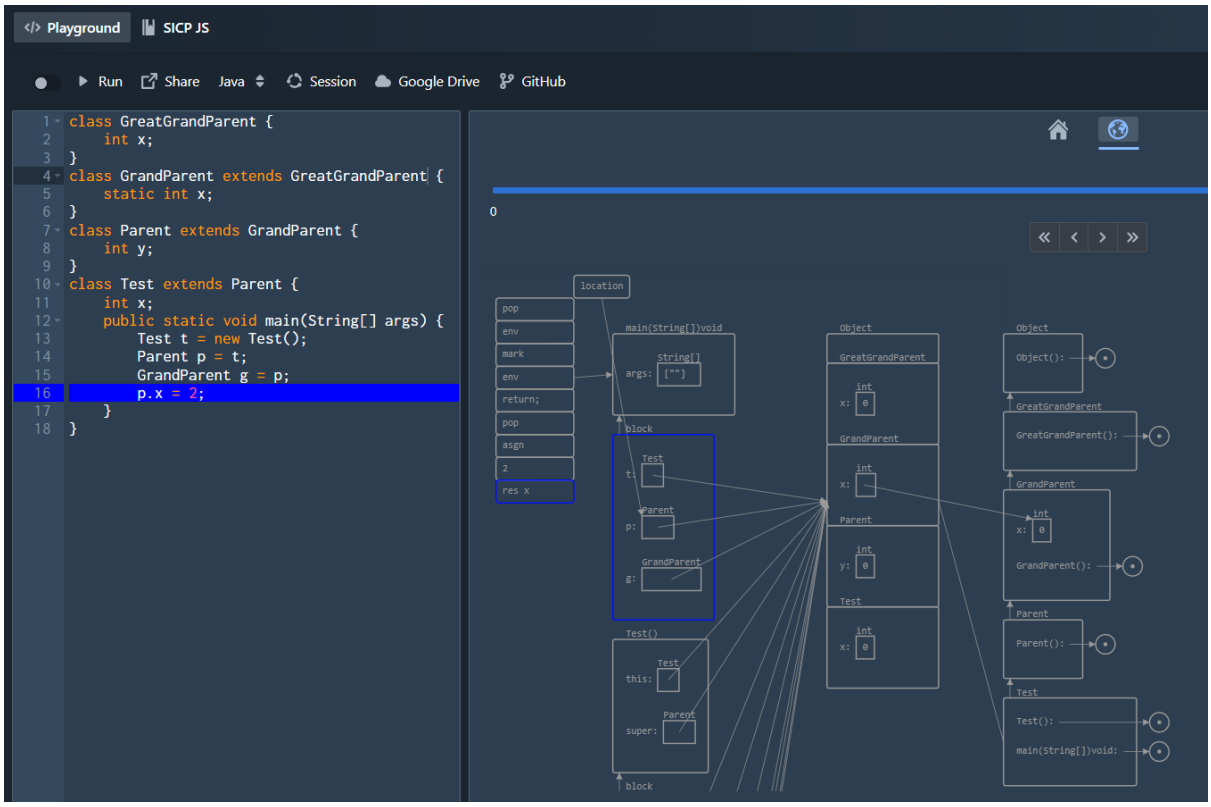
Prepending simple names in instance methods with this keyword

3.4.2 Field Access and Typing

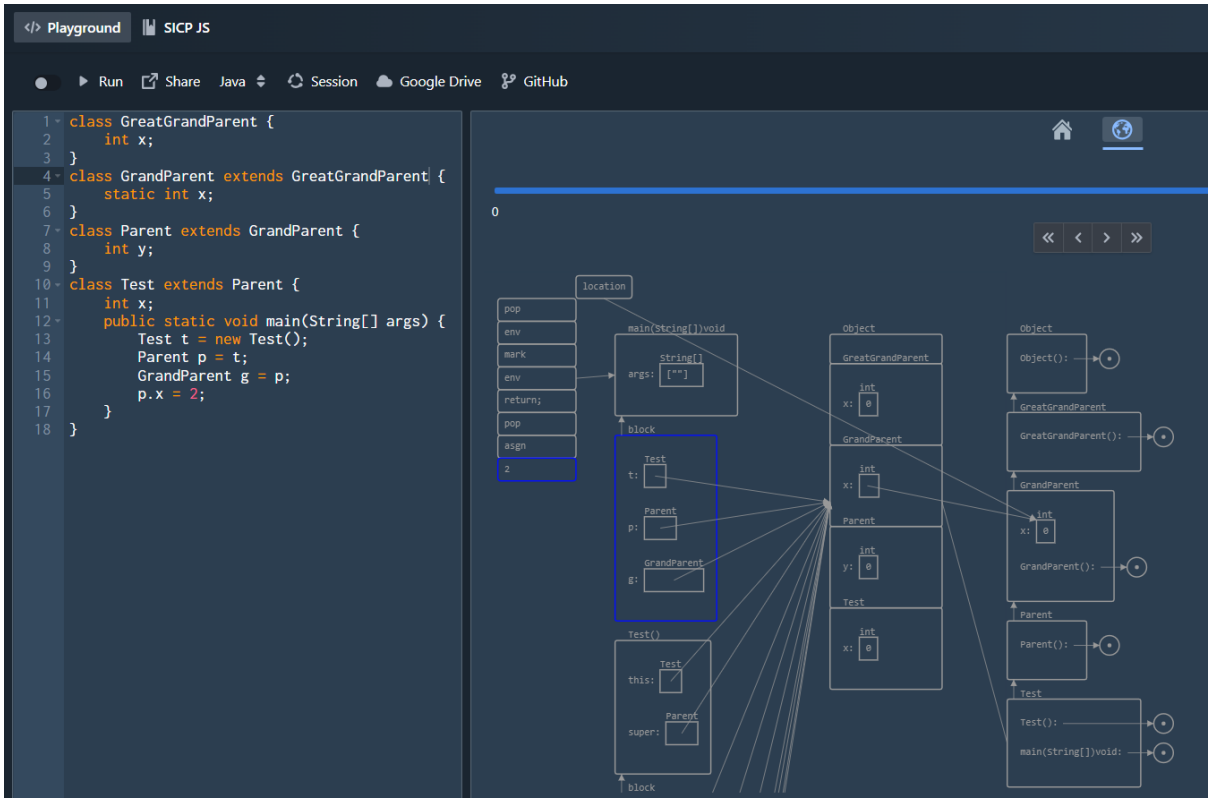
Classes define types and the subtyping relation introduced by inheritance makes field access more involved. The type of variable, which is explicitly annotated on top of the box, dictates

which frames the variable can access. For example, as p is of type Parent, attempting to access field x of variable p leads to looking up of name x from Parent frame of the object that variable p refers to onwards, where the name x is resolved to an inherited class field x.

Interestingly, such a behavior reflects the relationship between object-oriented programming and typing which should be a topic of discussion in class.



Type of variable p constrains the frame where name x is resolved.



Name p.x is resolved to the class field x of GrandParent class.

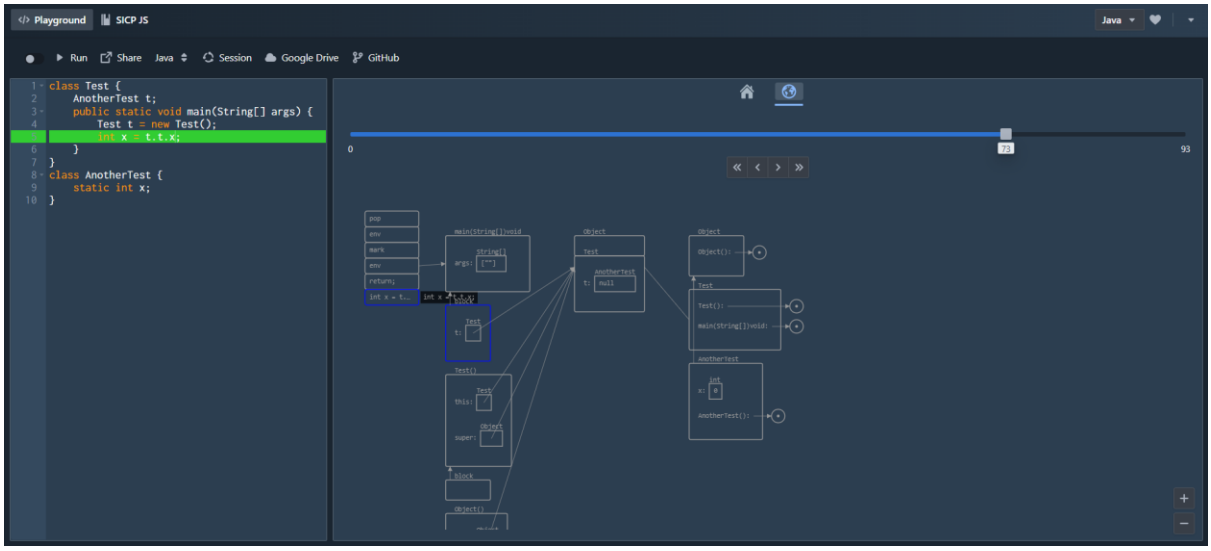
3.4.3 Location and Dereferencing

As we see that name resolution can be quite involved, the use of name to refer to location or content should be decoupled from the resolution process.

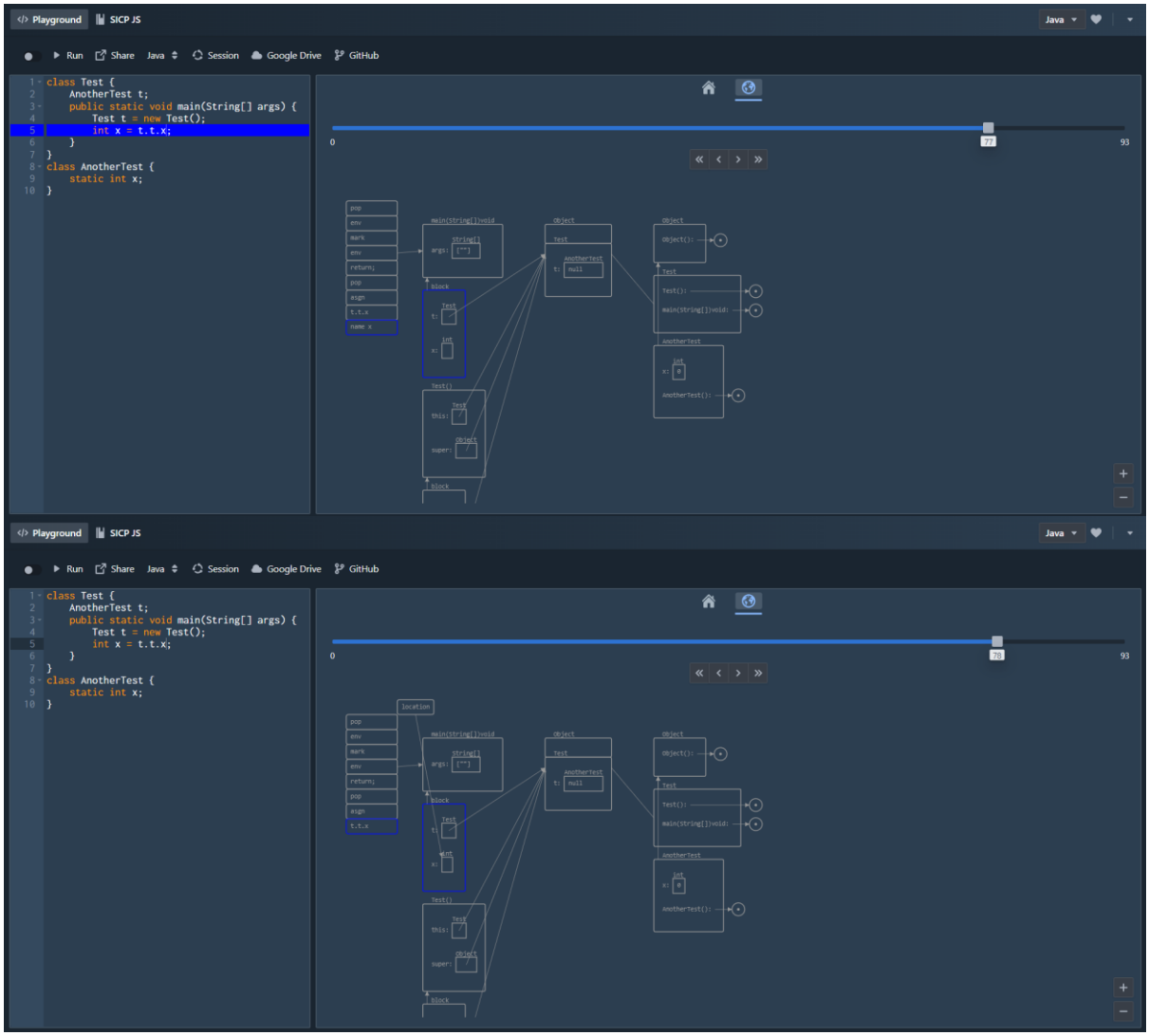
The notion of lvalue originated from C/C++ and appears in Java as well. Let's examine these concepts using Assignment. The name on the left hand side of an Assignment requires a location while names on the right hand side requires a dereferenced value.

The notion of location is more apparent in object-oriented languages with fields and accessing fields using qualified names. These names need to be resolved into locations accordingly before proceeding with further evaluations.

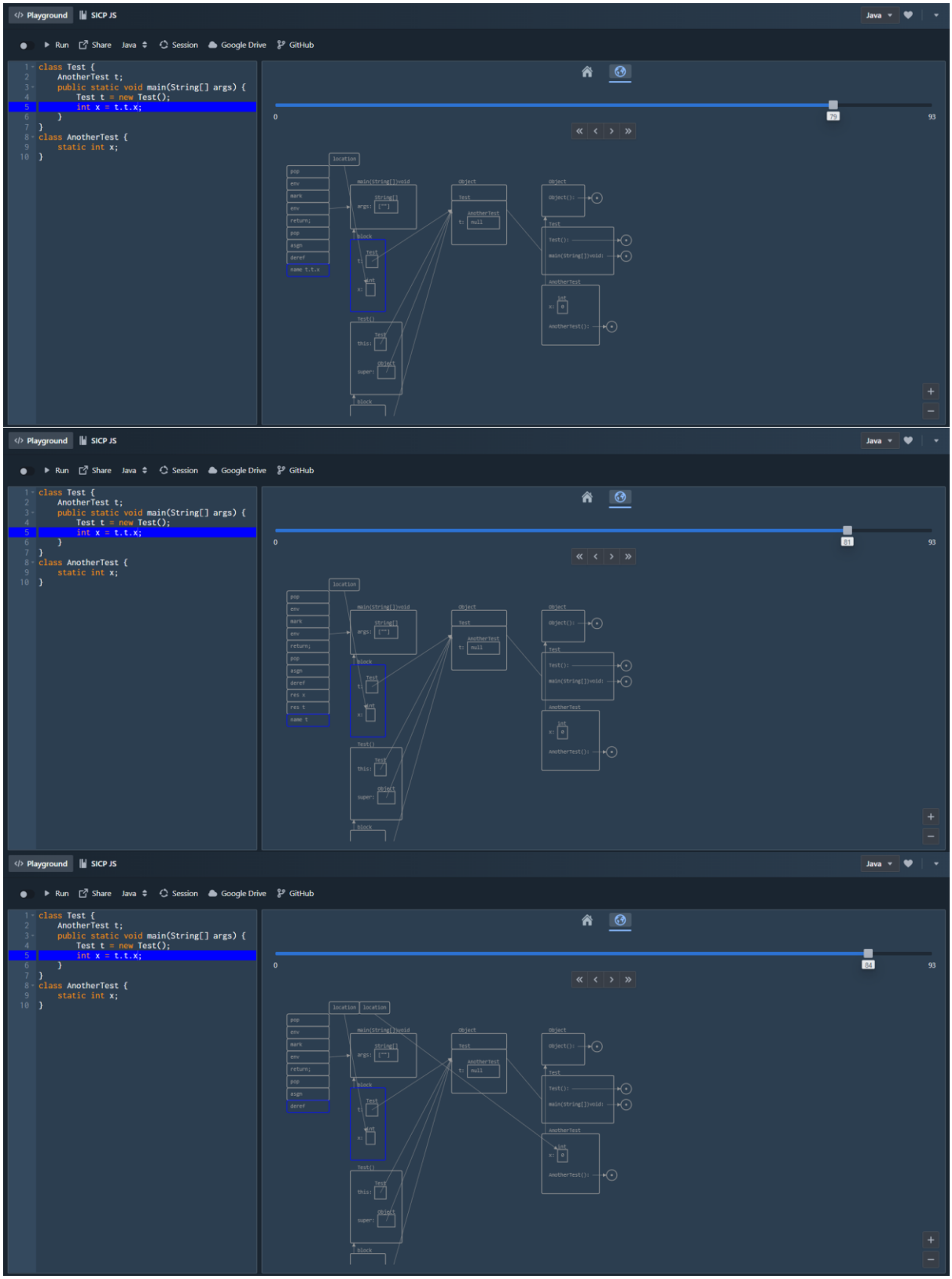
The motivation behind having a box for variables is to represent locations as boxes while dereferencing a name is analogous to taking the content of the box.



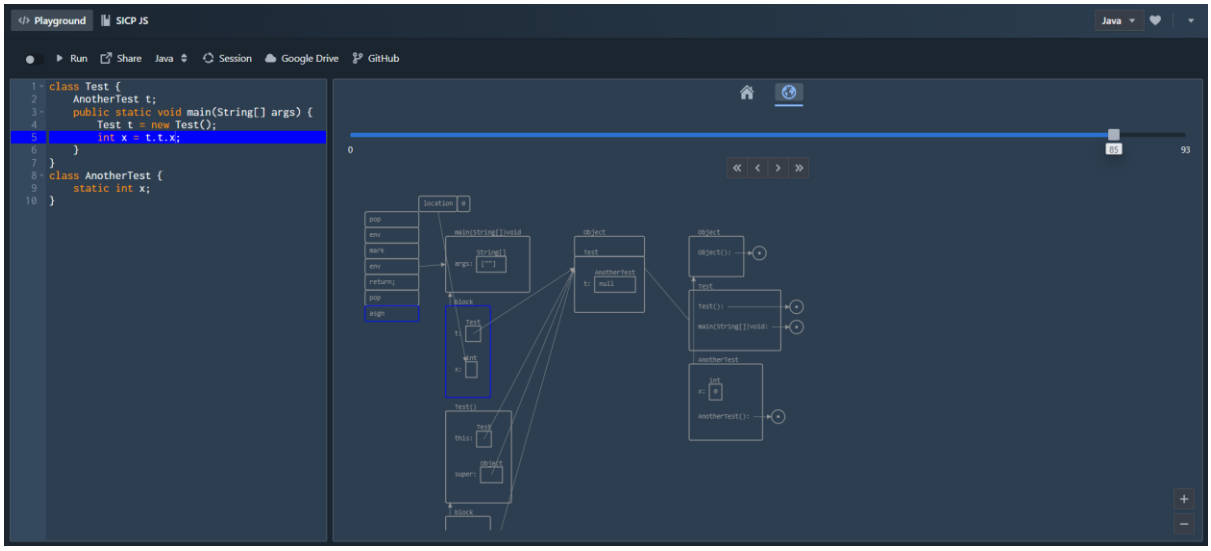
Evaluating assignment of value of field x of field t of local variable t to local variable x.



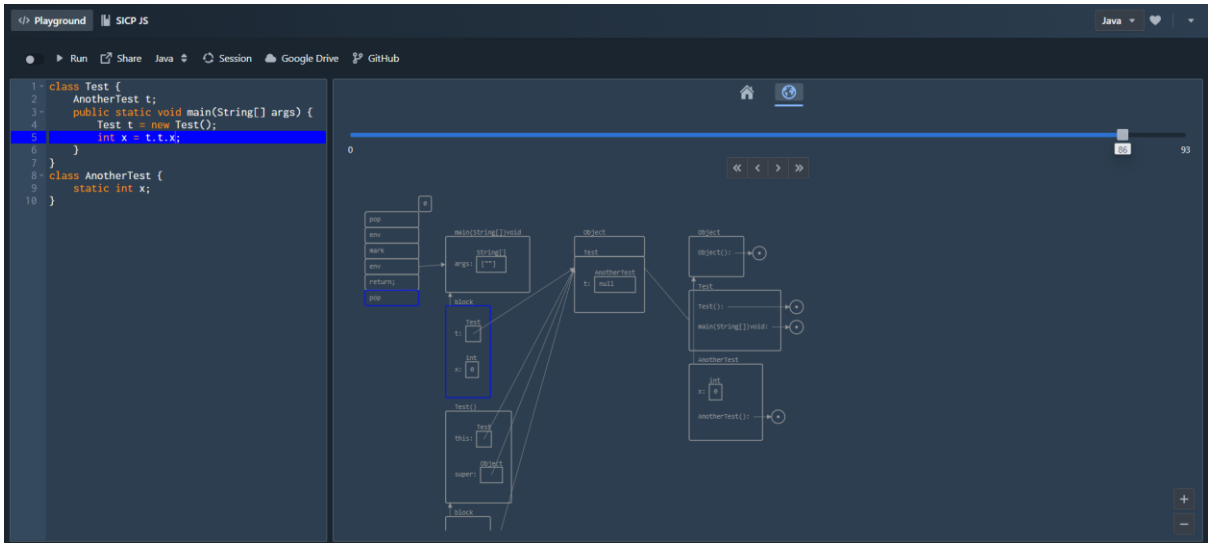
Name x is resolved to a location.



Name t.t.x is resolved to a location.



The location referred to by name t.t.x is dereferenced to get its value 0.



Assignment of 0 to location referred by name x.

3.5 “Desugarization”

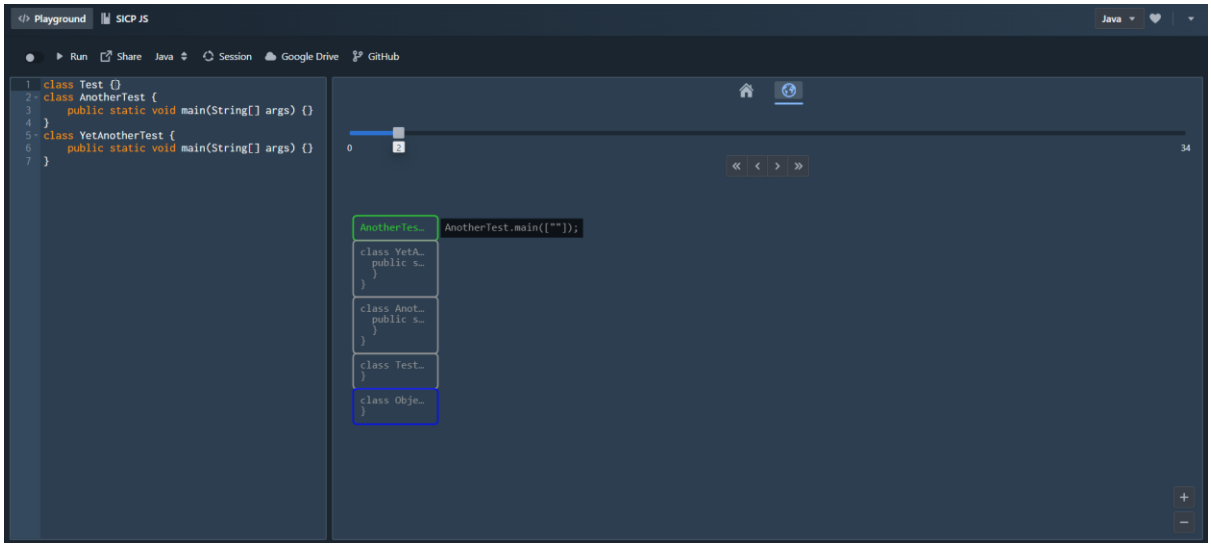
As object-oriented programming is very powerful, much structure is required to support the architecture. However, these structures contribute to verbosity and are often sugarized for the benefit of developers. Let’s desugarize these constructs one by one.

3.5.1 Initial Main Method Invocation

In the spirit of making things explicit, implicit constructs are explicitly visualized. The implicit constructs include the initial main method invocation, root Object class, default constructor,

Being designed as a compiled language in mind, Java is natively class based and all code is put into classes. The initial method to be invoked is the public static void main(String[] args) (James Gosling, 2023). Differing from the command line setup, this implementation invokes

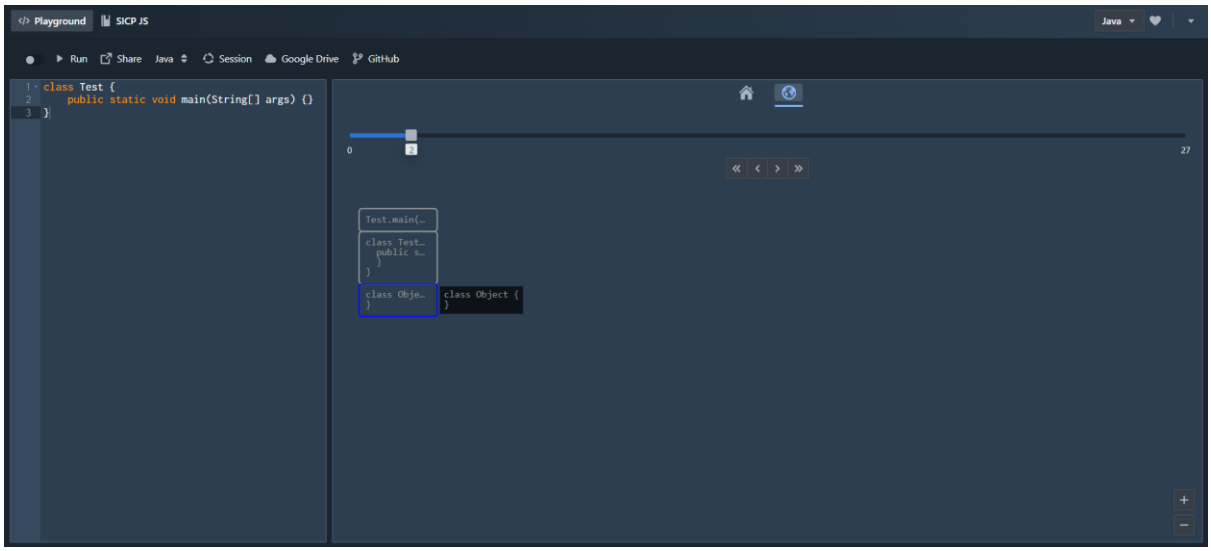
the main method of the first class that declares the main method in program order. Hence, in the spirit of making things explicit, a corresponding main method to be invoked is pushed onto the control explicitly.



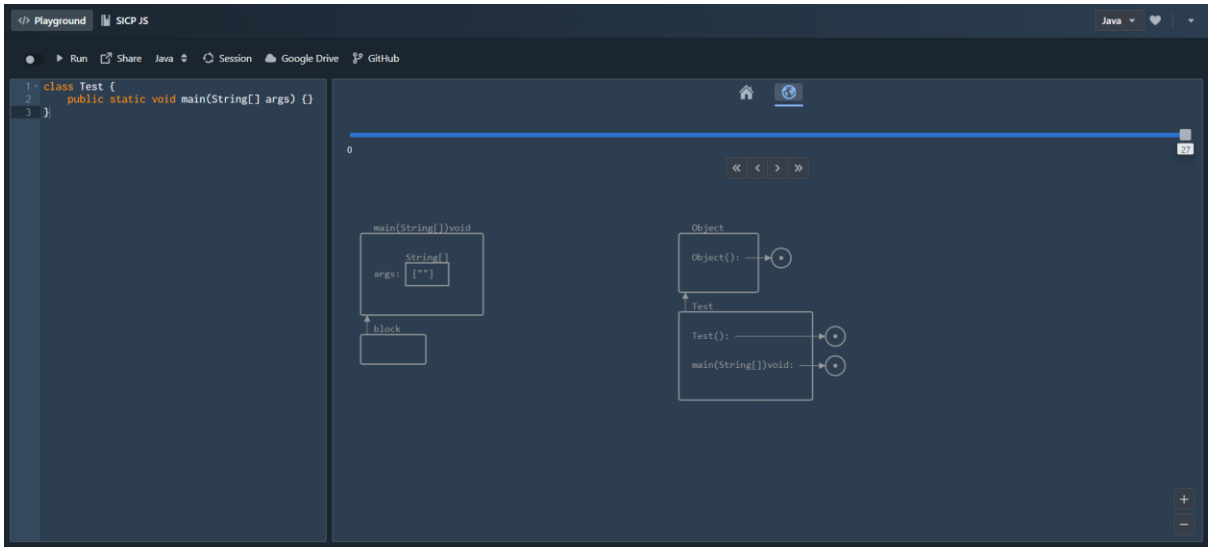
AnotherTest::main method invocation is pushed onto control to be evaluated.

3.5.2 Root Object Class

Every class that does not explicitly inherits any class implicitly inherited the root Object class (James Gosling, 2023). In the spirit of making things explicitly, the Object class is added into the user code, which will eventually get evaluated like any other user defined classes.



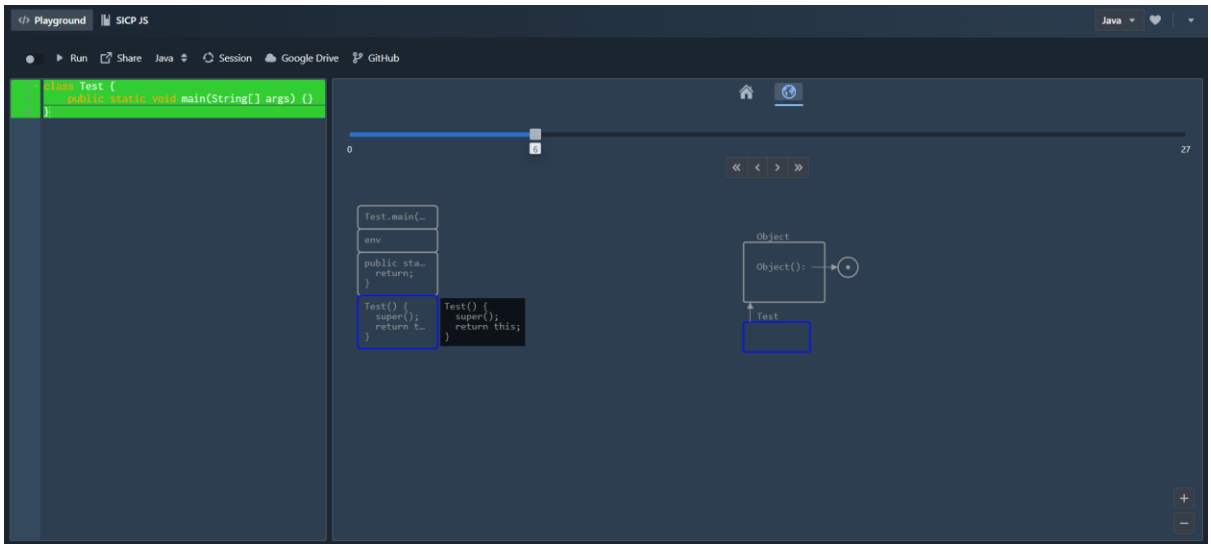
Root Object class declaration is pushed onto control to be evaluated.



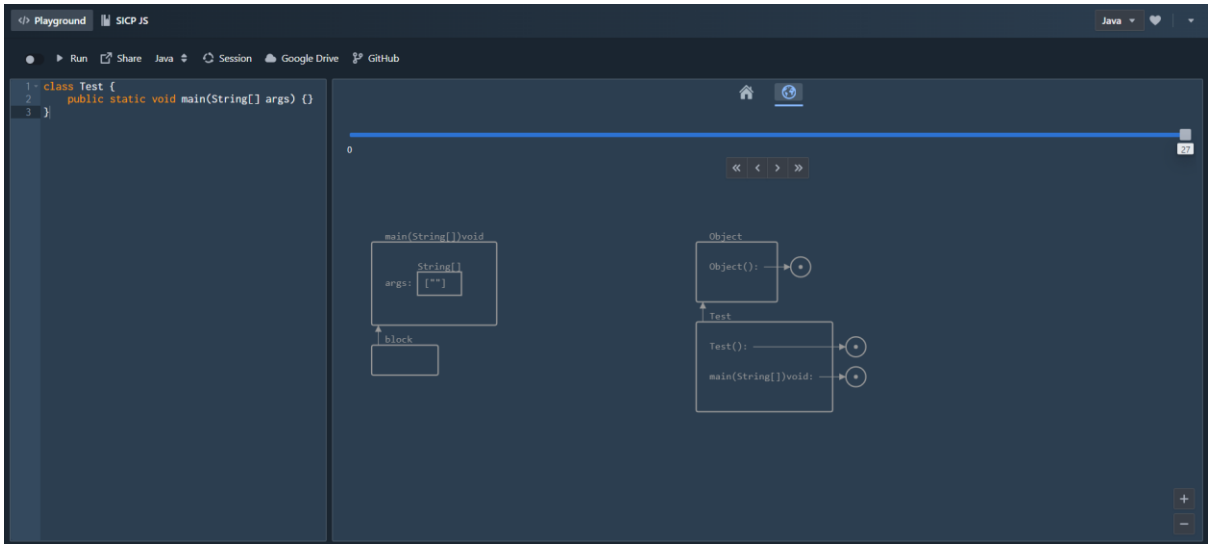
Class Test inherits from root Object class.

3.5.3 Constructor

Every class that does not explicitly define a default constructor implicitly defines a default constructor (James Gosling, 2023).



Default constructor declaration is pushed onto control to be evaluated.

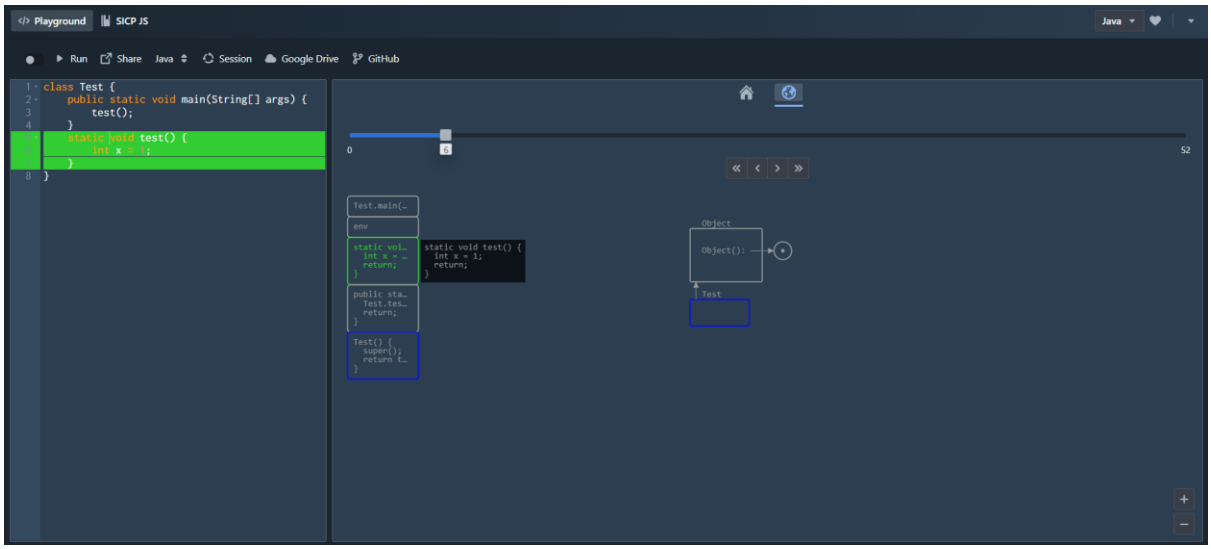


Class Test defines a default constructor in its frame.

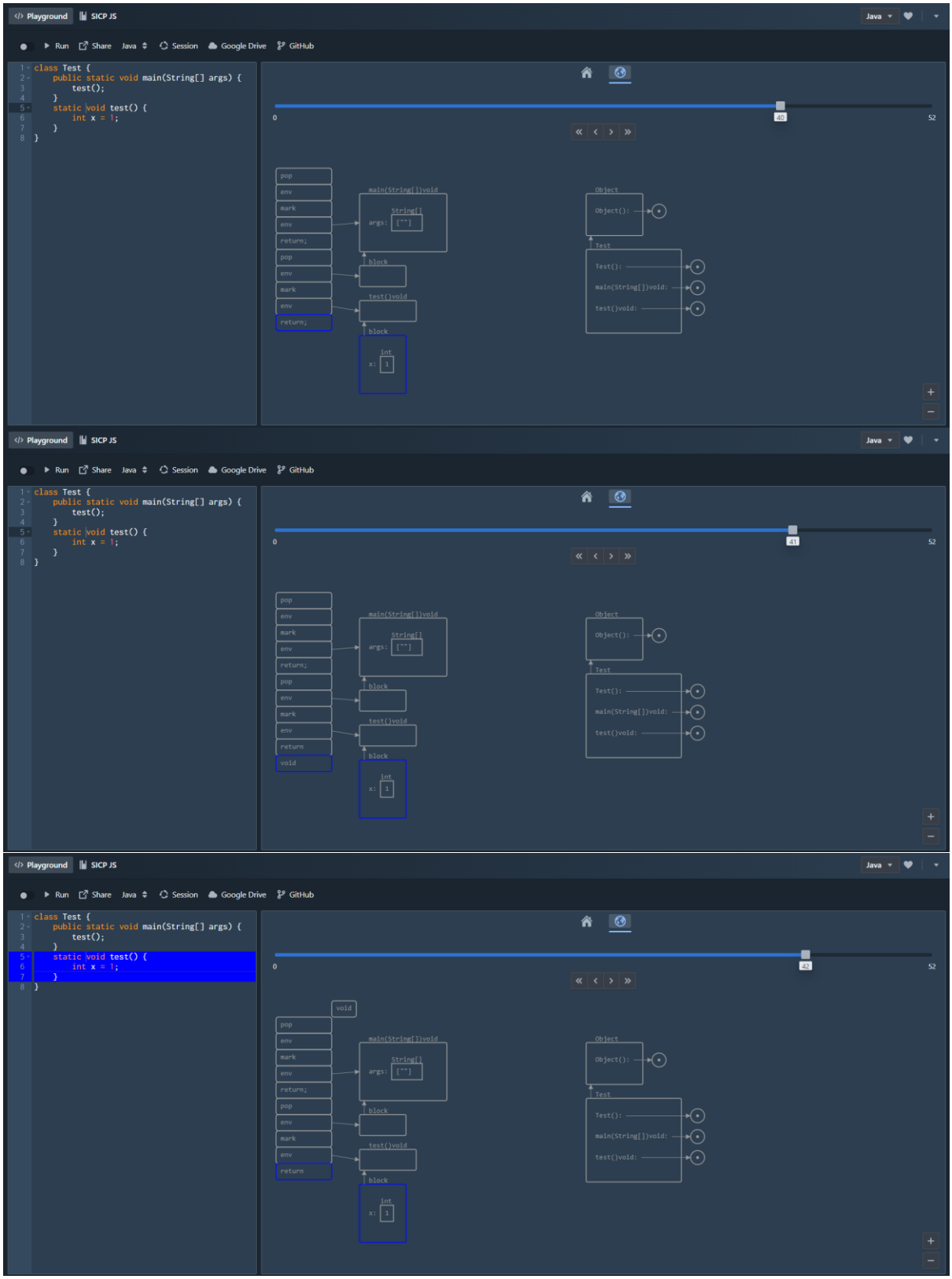
3.5.4 Return Statement

Method declarations with void return type does not require explicit return statements (James Gosling, 2023).

In addition to appending the implicit return statement, to ensure that every return statement returns a value, a special Void expression is returned for method declarations with void return type. Such augmentation unifies the handling of method invocation expression that always produces a value, and the value will be popped after.



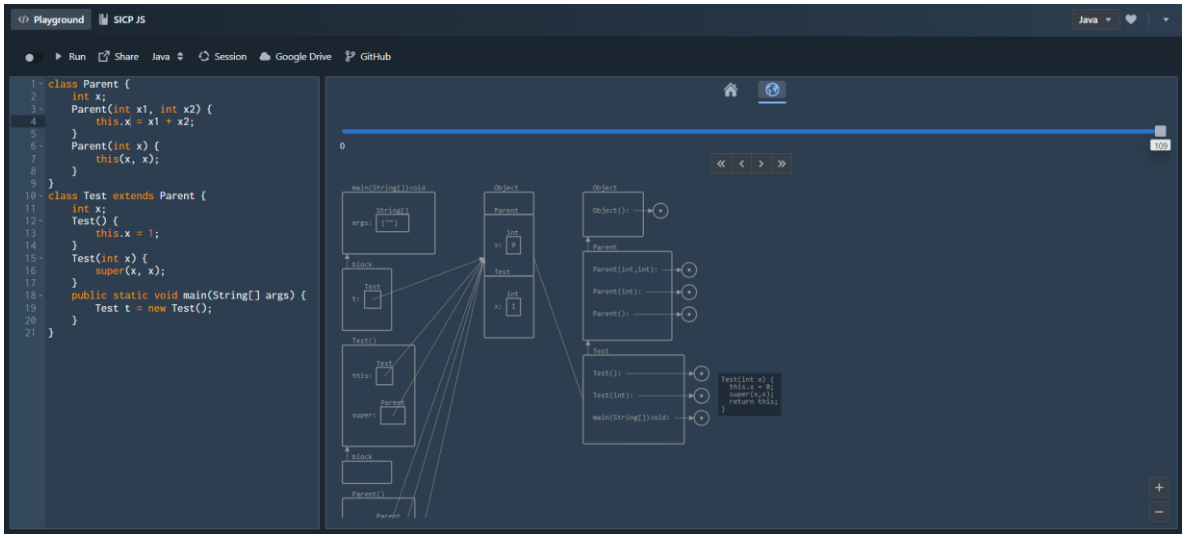
Empty return statement is appended to method body whose method return type is void.



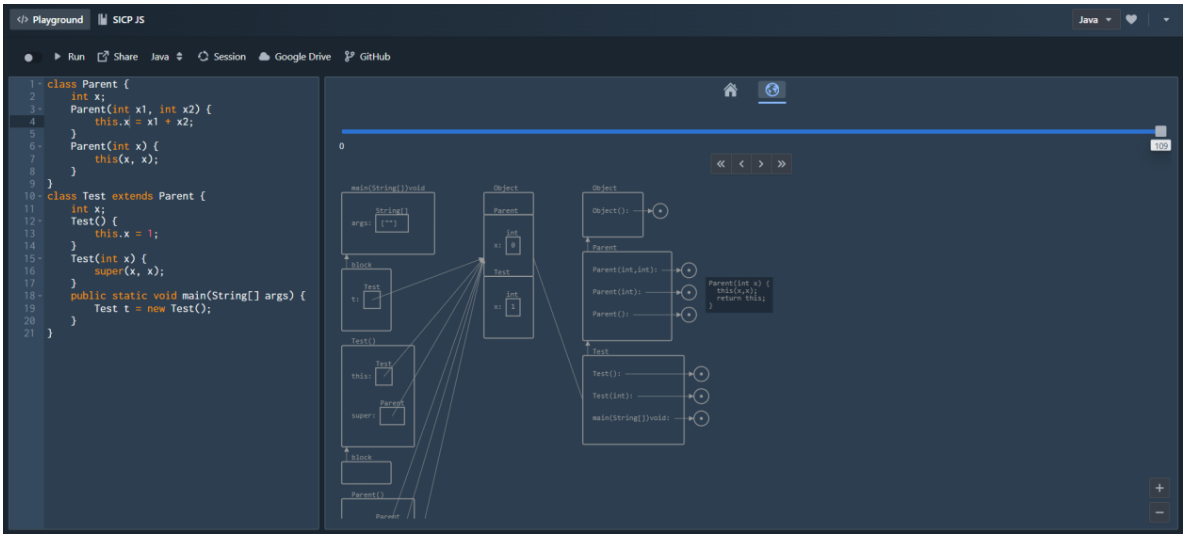
Empty return statement pushes special Void expression onto stash.

3.5.5 Superclass Constructor Invocation

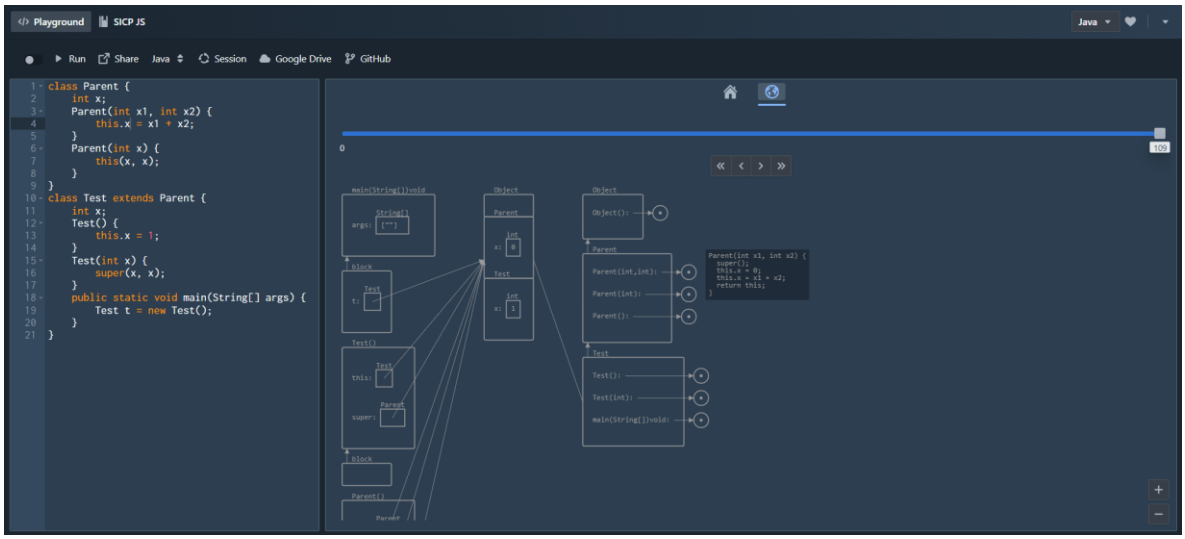
The superclass default constructor is invoked in constructors whose class inherits another class that do not invoke any explicit constructor (James Gosling, 2023).



No superclass default constructor invocation to Parent() as an invocation to another superclass constructor Parent(int, int) is present.



No superclass default constructor invocation to Object() as an alternate constructor invocation Parent(int, int) is present.



Superclass default constructor Object() invocation is inserted as no other explicit constructor invocation is present.

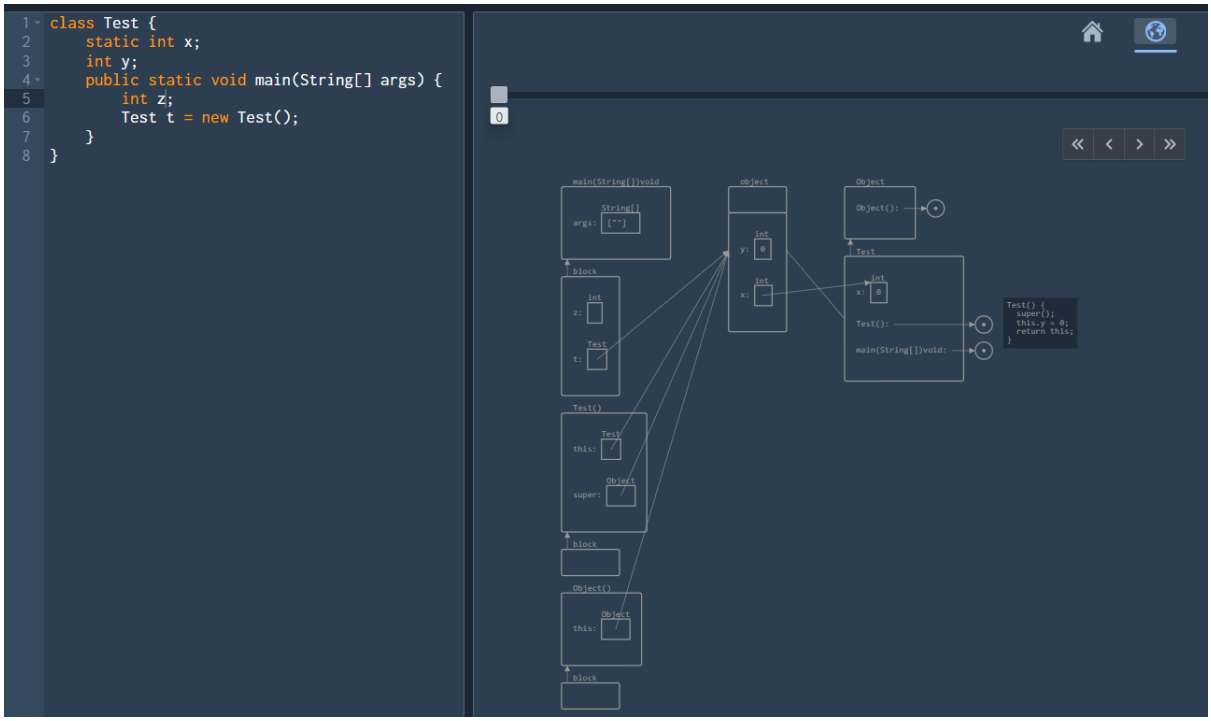
3.6 Variable Declaration and Assignment

Note that the term “variable” includes local variables and fields.

It is possible to declare local variables without initializing them, but fields are always initialized to their default values. To emphasize this distinction, local variable declarations with initializer are transformed into local variable declarations without initializer and assignments to the initializer value, while field declarations, with or without initializer, are evaluated straightaway.

Also, to make the initialization of fields to their default values explicit, class bodied are pre-processed to augment class field declarations without initializer to class field declarations with default value as initializer, and constructor bodies to prepend assignments of default values to instance fields.

Implementation wise, declaring a variable means allocating memory for the variable and assigning a value to a variable means writing to the memory. To provide a sufficient abstraction, the CSEC visualizer creates a box upon declaration and populating the box with a value during assignment.



The box for local variable z is empty while the boxes for class field x and instance field y are initialized to default values.

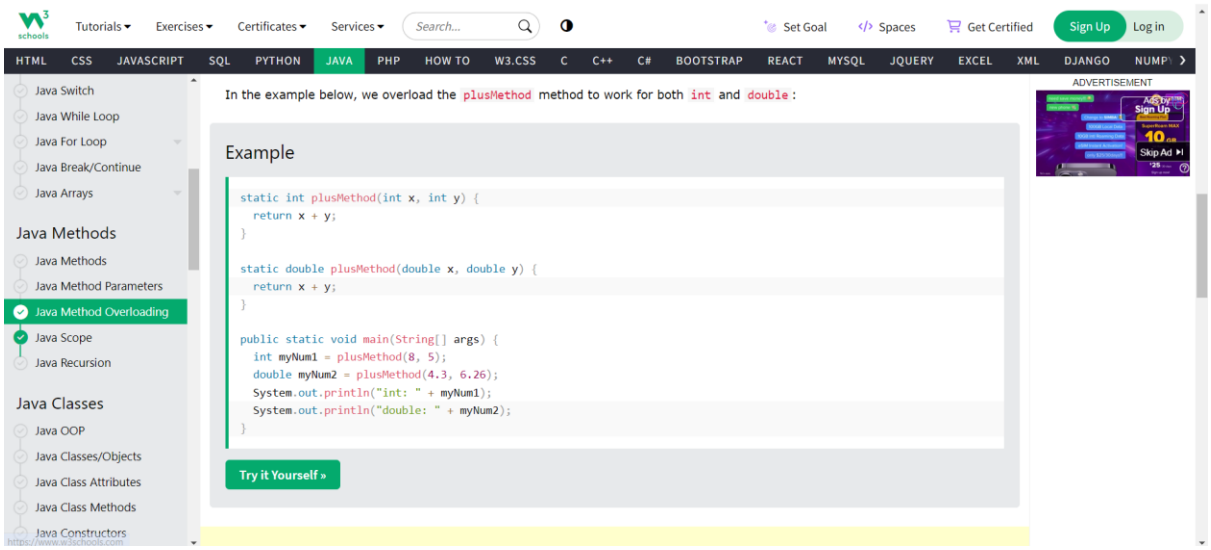
3.7 Method Resolution

Having seen that name resolution can be quite complicated, method resolution is unfortunately more complicated. However, understanding method resolution in depth sparks discussions

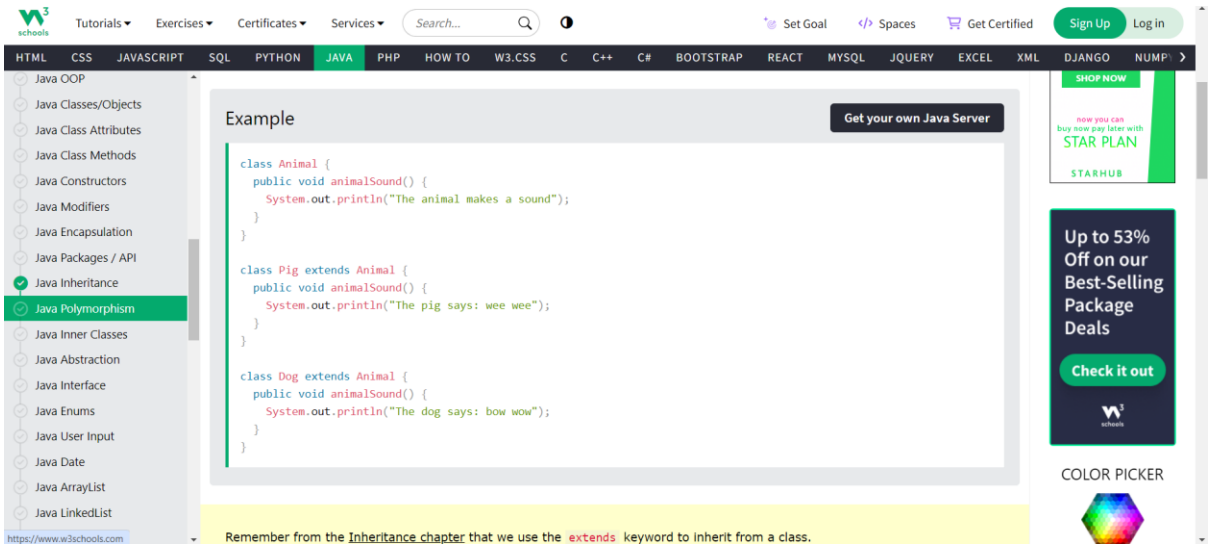
regarding interesting topics such as typing and dynamic binding which leads to polymorphism. There are a total of three parts in method invocation: two parts of method resolution: method overloading resolution, method overriding resolution, and method invocation.

Method overloading resolution and method overriding resolution is often discussed separately, where the formal is discussed as early as data types are introduced while the latter is introduced only after inheritance and polymorphism are introduced. However, the link between these two resolution steps is missing, which is crucial to understand the entire method resolution process. We shall see below an example where this missing link affects our understanding of the method resolution process.

Also, the reliance on intuition when resolving method overriding when it comes to some possibly simple but uncanny programs. While intuition is a good starting point to introduce the new concept of method overriding, a more thorough formalization of the method overriding resolution should be considered. We shall inspect one such program below.



Separate discussions on method overloading and method overriding (W3Schools, n.d.).



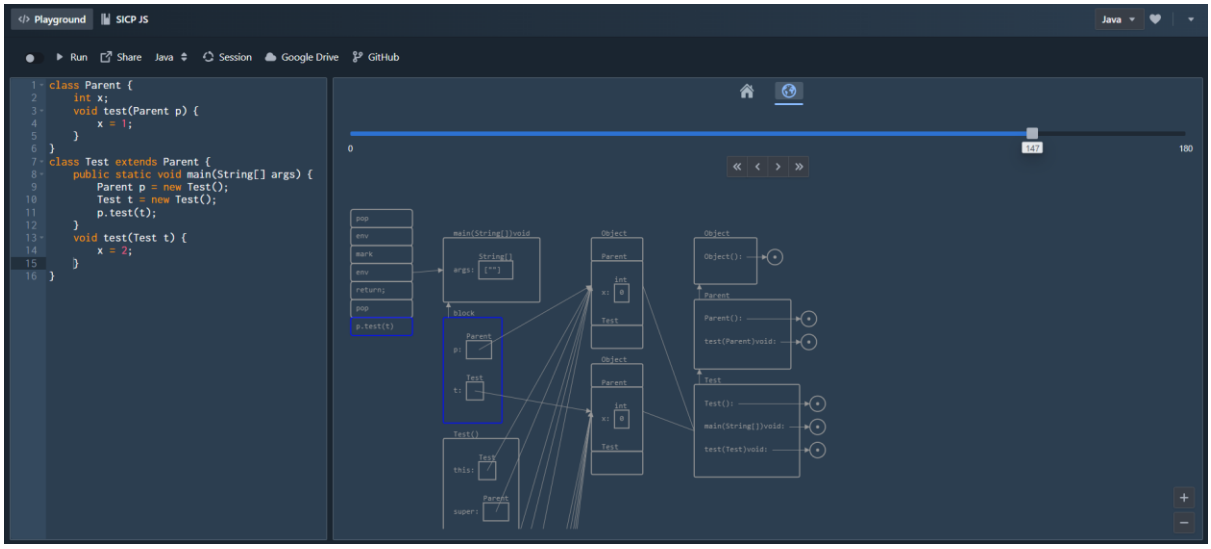
Use of intuition when introducing method overriding (W3Schools, n.d.).

3.7.1 The Two Stages of Method Resolution and Method Invocation

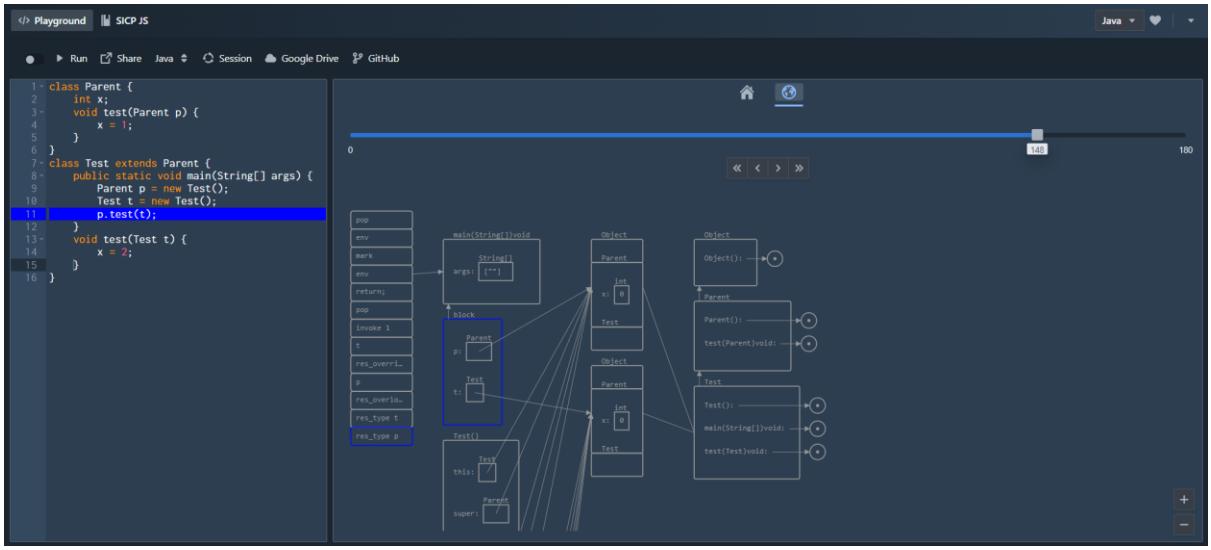
As method overloading is allowed in Java, i.e., same name different parameter types, typing is prominent in method overloading resolution. Before even resolving the argument types against the parameter types, the type of the target restricted the region where a method is searched in, like how the type of a variable restricts which object frame an instance field is looked up in. Here target refers to the qualifier. Recall that simple method names are pre-processed to prepend a qualifier depending on whether the name is in a class or instance method. The emphasis on typing is portrayed using the RES_TYPE instruction as well as the type-annotated environment.

Next, method overriding takes in the resolved overloaded method and the target object, and continues to resolve the method to be invoked. It is important to note that method resolution is done in stages and the method overriding stage depends on the method overloading stage.

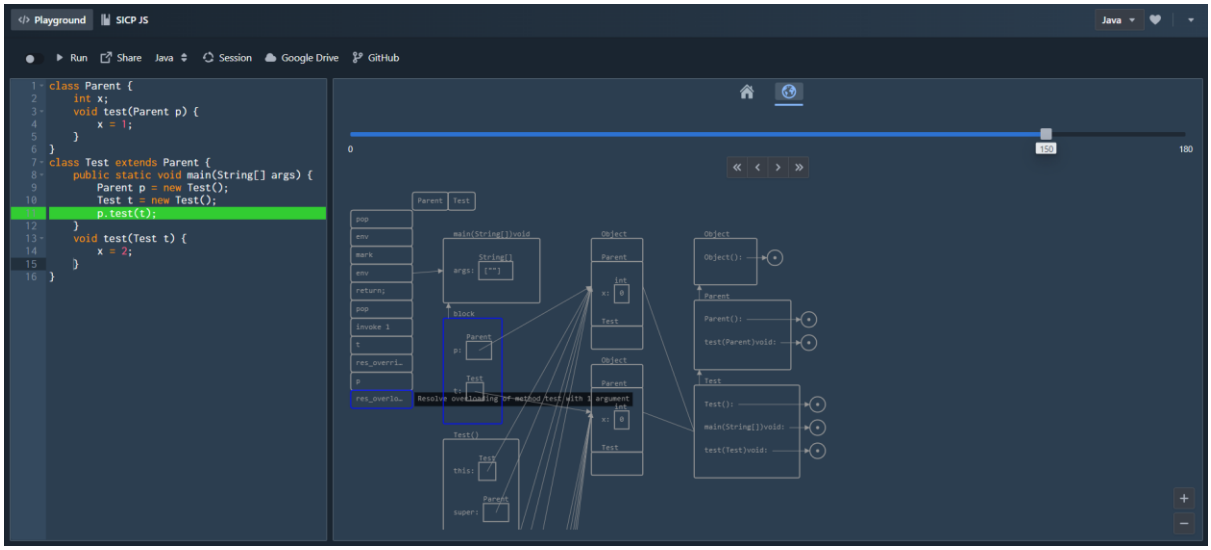
The final stage is invocation, where arguments are evaluated, and the current environment is extended with a frame that binds formal parameters to the respective argument values.



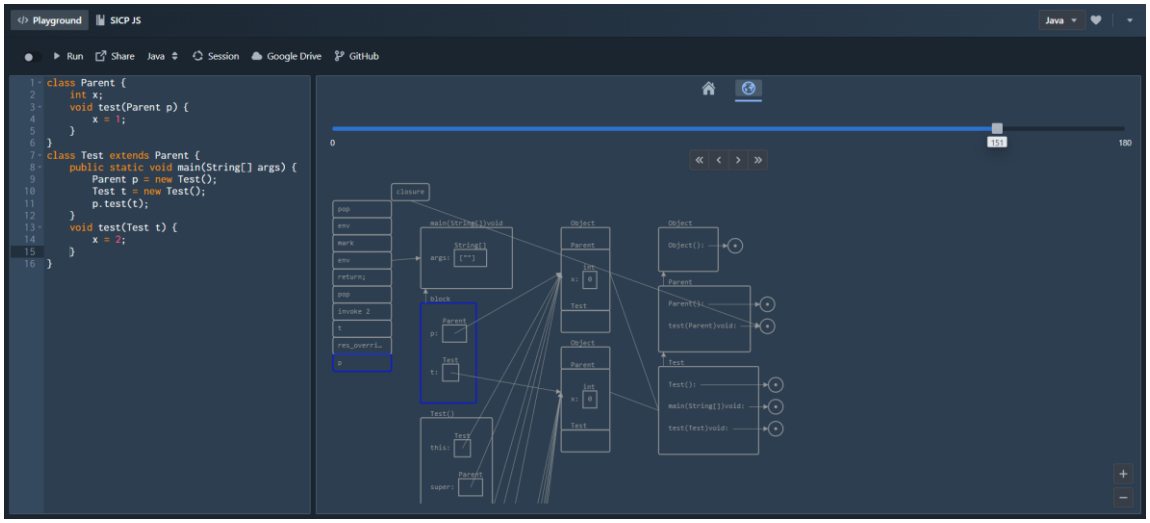
Evaluating method invocation p.test(t).



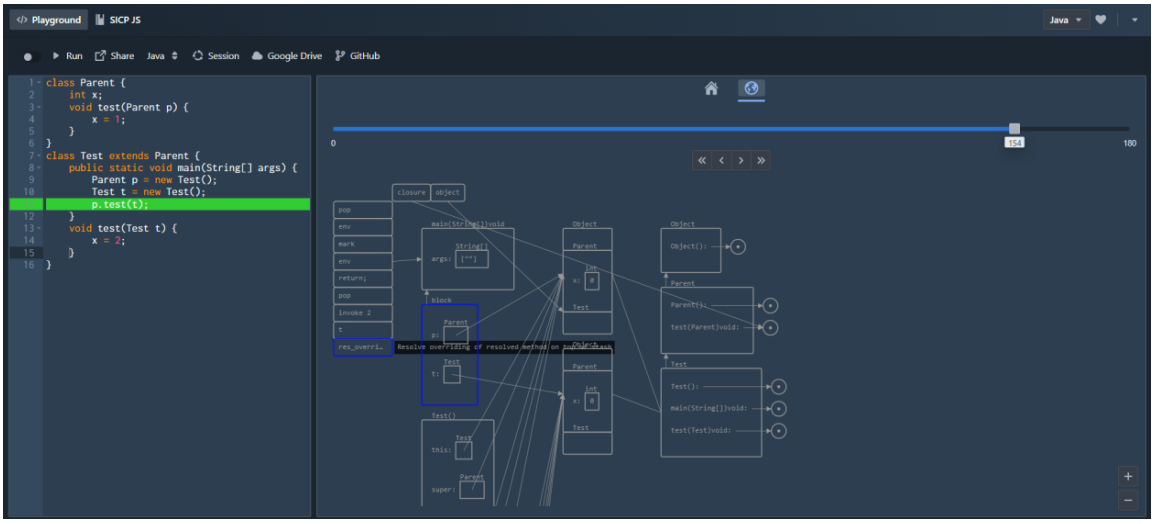
The two stages of method resolution and method invocation are shown consecutively on control.



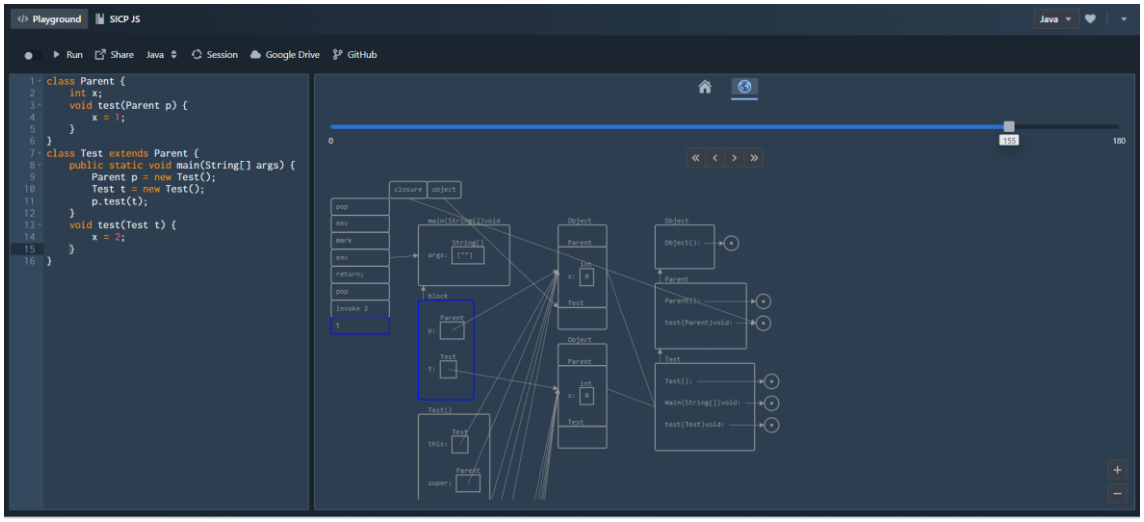
Types are used in method overloading resolution.



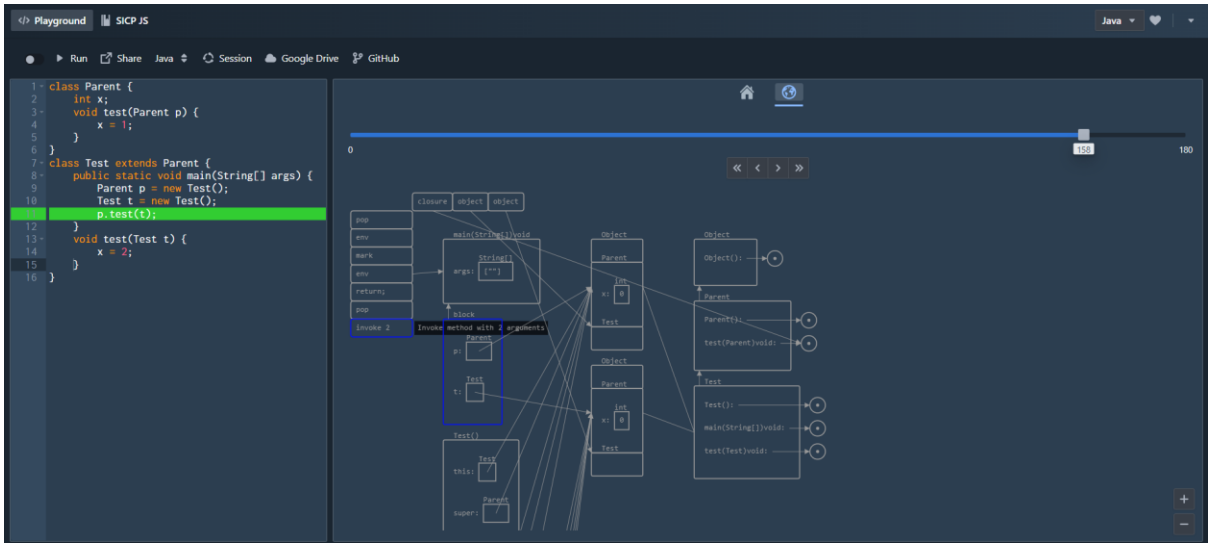
Method overloading resolved method is pushed onto stash to be used in method overriding resolution next and INVOCATION on control is modified to 2 arguments as the resolved method is an instance method.



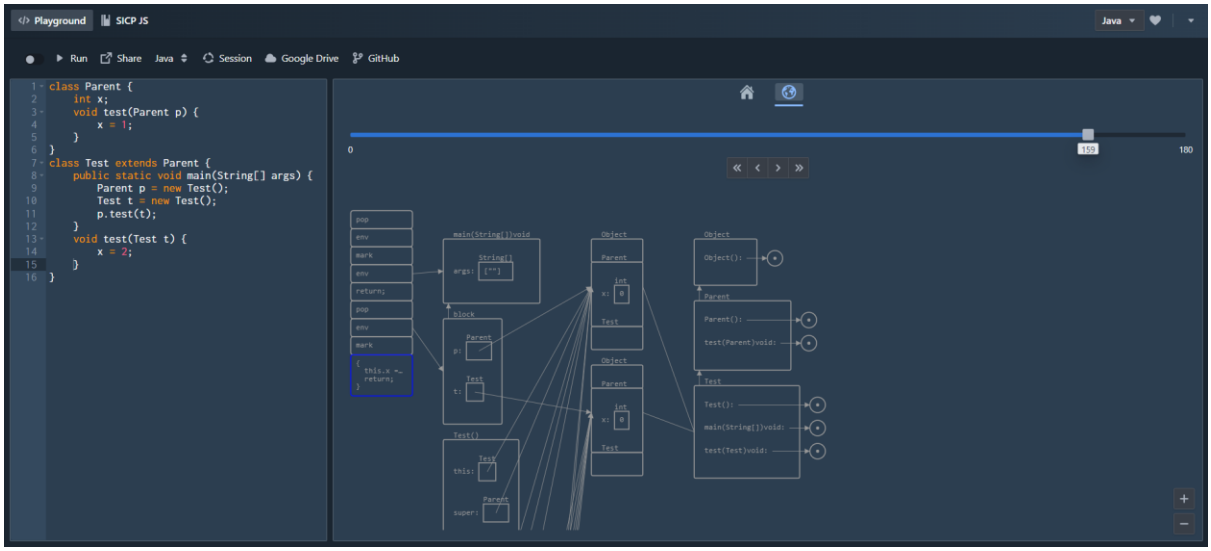
Method overriding resolution takes in resolved method and target object.



Method overriding resolved method is the same as method overloading resolved method due to the absence of an overridden method of the same method signature as the method overloading resolved method.



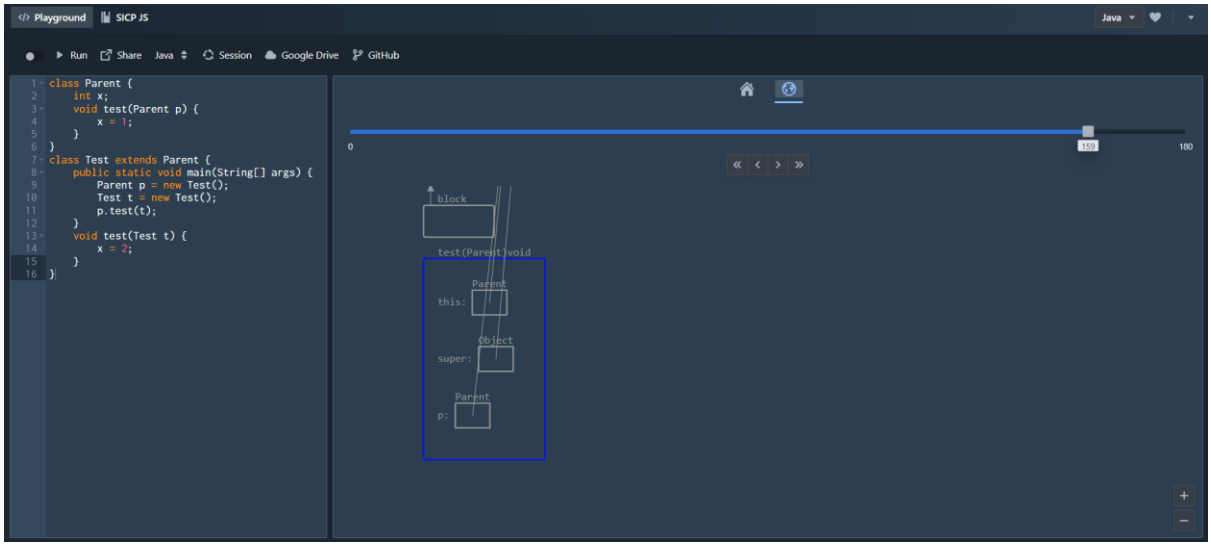
Only arguments are evaluated now as target has been evaluated before method overriding resolution so just push onto stash after use in method overriding resolution.



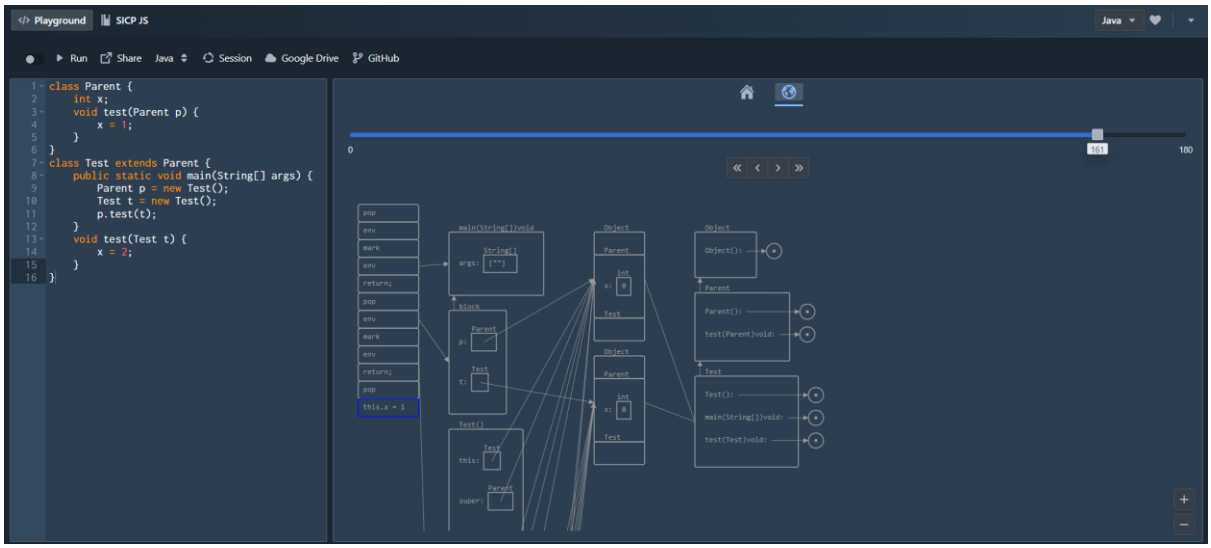
The correct method is invoked at the end.

3.7.2 this is Yet-Another-Variable

The target object is also passed as an implicit argument along with the other arguments and is bound to the name this and super. Note that although this and super binds to the same object, they have differing types. Such a treatment of the this and super keyword simplifies the resolution logic.



Implicit arguments this and super.



this is yet-another-variable and name resolution proceeds as usual.

3.7.3 Hiding

Hiding is a situation where the method of the same signature is overridden in the subclass. This scenario is encoded into the method overriding resolution process.

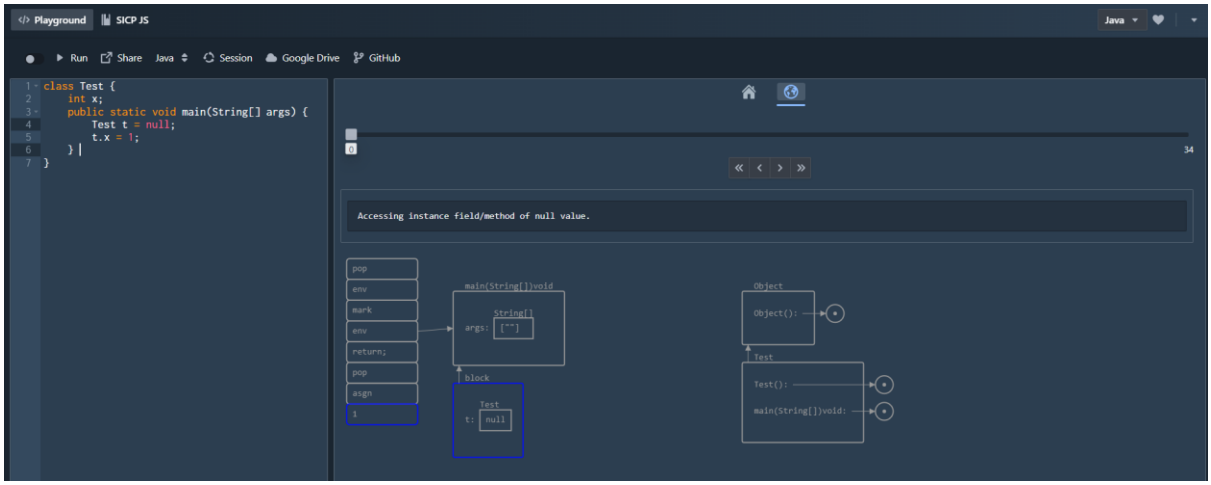
3.8 Assumption

The Java CSEC Machine expects syntactically valid and well-typed Java programs as input. Such an assumption is justifiable as a notional machine is purposed to explain program execution.

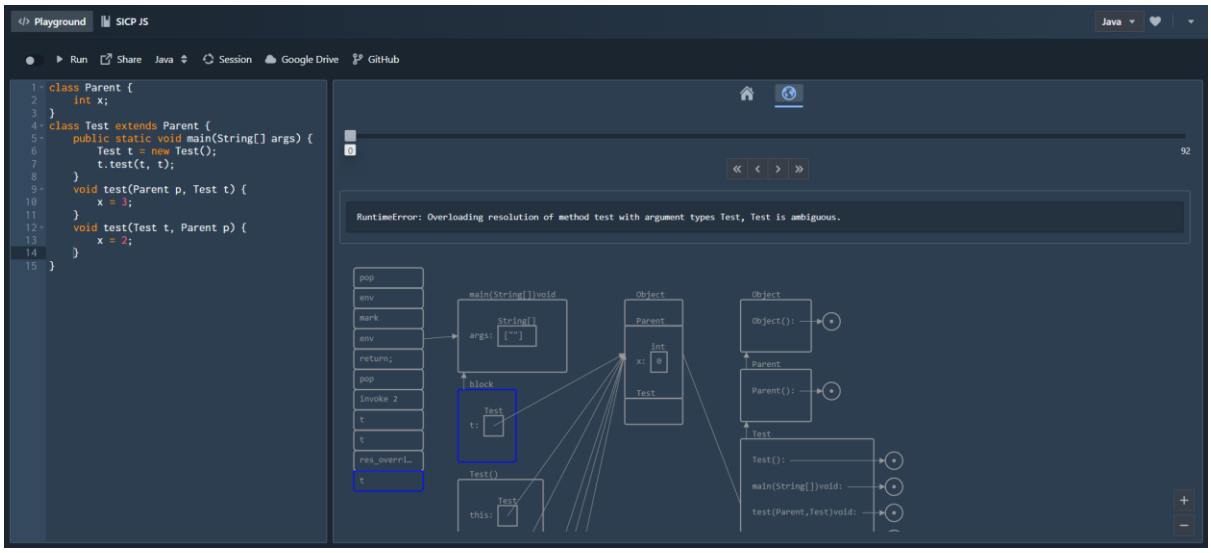
However, there may still be “compilation” and runtime errors in a syntactically valid and well-typed program, e.g., method overloading resolution error, NullPointerException, etc.

3.8.1 CSEC Machine as a Debugging Tool

The machine can get stuck – it's a feature, not a bug! Such a feature provides users a glimpse into the source of error as well as a trace leading up to the error being thrown.



NullPointerException runtime error.

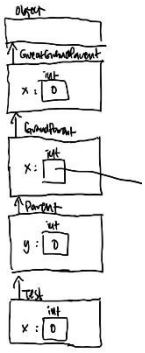


Method overloading resolution "compilation" error.

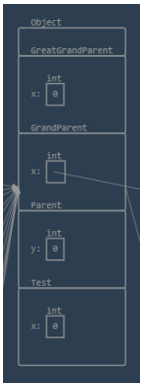
4 Design Decisions

4.1 Objects & Class Field References

An object is internally represented by multiple frames. Like classes, the initial object representation of chain of frames are chained by arrows. However, such a representation may provide an impression that multiple objects are created which is an inaccurate mental model. In the end, these frames are decided to be bundled together to signify that only 1 object is created with multiple frames represented internally.

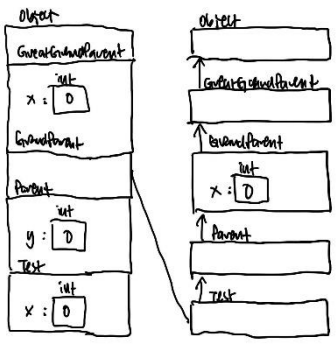


Object as chain of frames.

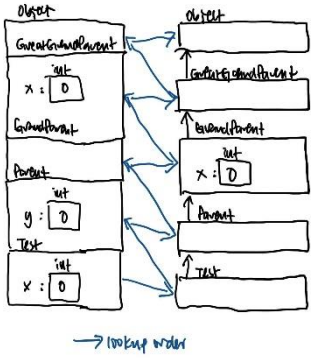


Object as multiple frames.

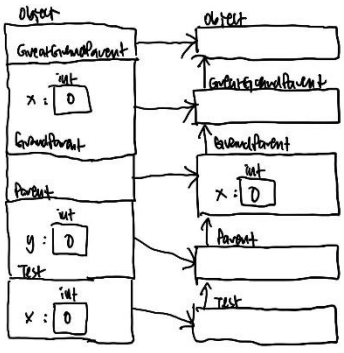
Class field references are introduced to better portray the name look-up order and reduce the complexity of name look up in implementation. As class fields exist in class frames and instance fields exist in object frames but the look up order is going through both instance and class fields in one class before moving to the superclass.



Initial Object Representation.

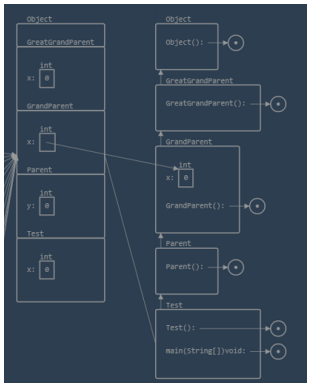


Look-up Order.

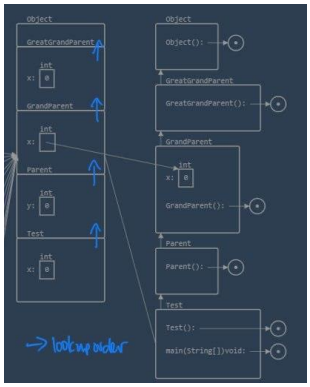


Naïve Attempt.

The original visualization breaks the look up flow as there is no sense of direction. There is also a naïve attempt to portray the look-up order but just does make sense and may potentially cause more confusion.



Current Object Representation



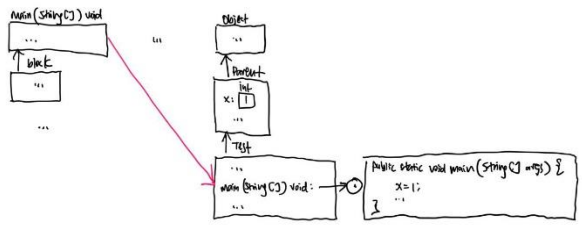
Look-up Order

Class field references was decided on as it seems to be an elegant solution by far, although the object representation as chain of frames may better portray the lookup order due to the presence of arrows. Point to note that it is also a bad impression to have the class field in the object frame as well since class fields are semantically to be shared among all objects.

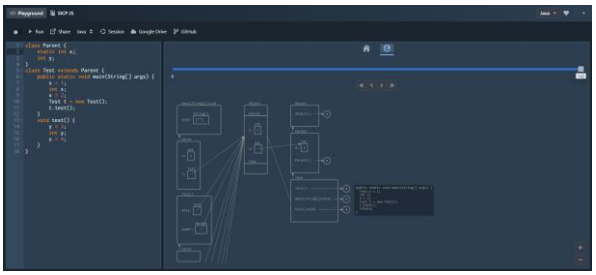
4.2 Augmenting Lexical Scoping

With simple names being possibly used to refer to fields, the intuitive solution would be for method frames to extend from objects or classes – instance method frames extend from objects whereas class method frames extend from classes. While this approach makes sense, the resolution is not as elegant as the current proposal where the emphasis on the new concept of field access being discussed is blurred with lexical scoping of local variables.

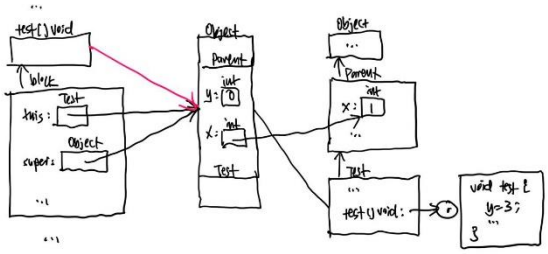
Also, the inheritance relation is not encoded into scoping, so extending environment does not mean enclosing environment. The specification states that the scope of a top-level class or interface is all class and interface declarations in the package in which the top-level class or interface is declared (James Gosling, 2023). However, the previous design goes against the language specification, which is not ideal, and perhaps even dangerous as it will result in a wrong mental model.



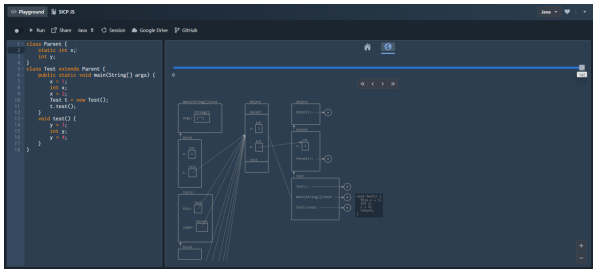
Without class name qualification.



With class name qualification.



Without this keyword qualification.



With this keyword qualification,

Note that for the case of instance methods, the arrow from the method frame to the object is redundant as we are not exploiting the presence of the "this" variable.

4.3 Syntactic Sugar VS Verbosity

Syntactic sugar hinders understanding and verbosity is necessary to depict the full picture. Object-oriented programming is indeed very powerful and such power is enabled via strong underlying structures. However, these structures are hidden from view as they are viewed as verbose during actual development but should be made explicit in learning.

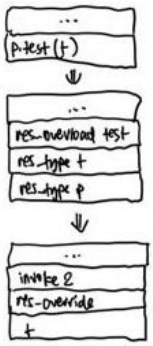
4.4 Location and Dereferencing

While it may be more verbose to dereference right hand side names which has a higher frequency than indicating left hand side names as lvalues which has a lower frequency, the former is preferred as the idea of a name referencing to a location and dereferencing whenever required is more elegant than the latter as dereferencing builds on the name resolution, which promotes reuse.

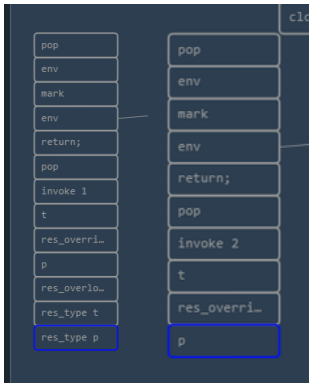
4.5 Method Resolution

As the second stage method overriding resolution depends on the result of the first stage method overloading resolution, and the passing of the target object as the implicit argument is dependent on whether the resolved method is an instance method, a previous attempt of unfolding the stages incrementally was adopted. However, such an approach breaks the whole thought process as subsequent stages seem to appear from nowhere.

The current implementation laid out the steps required right away and modifies the control stack should there be a need. This approach is straightforward as the unfolding is a constant regardless of whether the resolved method to be invoked is an instance or class method; steps not applicable just get skipped.



Dynamic unfolding but no control modification.



Static unfolding but control modified.

5 Implementation

5.1 Choice of Data Structures

As both control and stash are stacks, a Stack class is implemented with methods like push() and pop().

Environment is fundamentally made up of frames. Frames are TypeScript Maps instead of plain old JavaScript objects. Such a choice is made to prevent any confusion between predefined interpreting language object fields and interpreted language object fields.

5.1.1 Name Clash

It is possible for variables, methods, and constructors to have the same name and class frames to have all these constructs. As frames are represented by TypeScript Map, an elegant way to avoid name clashes is by taking the descriptor as the name of method and constructor. We can exploit the fact that parentheses are syntactically not allowed in variable identifiers and constructors do not have return type.

However, method overloading and constructor overloading are allowed, i.e., same name different parameter types. While this is a nice simple solution, looking up of method and constructor names take linear time rather than constant time. Note that looking up of variable names still take constant time.

To leverage implementing language constructs, TypeScript Map is chosen over plain old JavaScript objects, as per the Source 4 CSE Machine implementation. The need to use hasOwnProperty() method on these objects, to determine the existence of a name in the current frame exposes the structure of the underlying implementing language constructs. As this implementation could potentially be used as a sample implementation to further consolidate student's understanding of the concept being taught, as language implementers, we could consider the correspondence between teaching concepts and implementation structures.

In addition, TypeScript Maps allow use of non-string keys, which could potentially be exploited to improve look up time of methods and constructors.

5.2 Drawing Algorithm

The drawing algorithm is relatively straightforward, and the most important consideration would be the drawing order. Recall that the machine is composed of 3 components, i.e., control, stash, and environment. Visually, environment is broken down into 3 sections, i.e., method frames, objects and classes. The drawing order depends on references between components.

Since both items on control and stash reference things in the environment, the environment is drawn first. Within the environment, the sections are drawn from left to right as the largest width of the method frames determine the starting x coordinate of the objects section, ditto for classes.

Each java-slang component has a corresponding frontend component. The parent component is responsible to do the calculation for its subcomponents. Subcomponents are constructed and configured in the constructor of the parent component. Cascading draw methods are called after everything components are configured to the correct position and size.

All control items have fixed width and text exceeding the width will be cut and happened with eclipses, and the fill text will be shown in the tooltip.

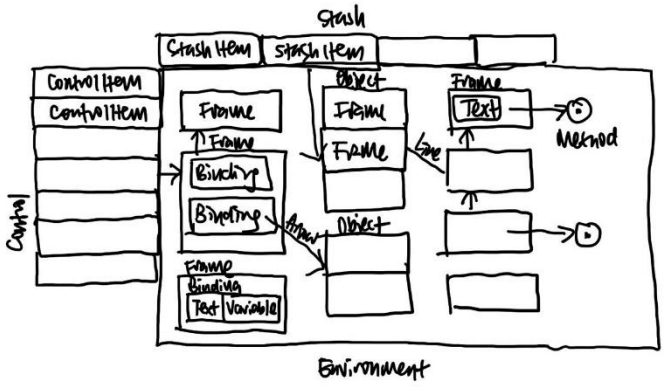


Diagram Layout

While there are existing work on more sophisticated drawing algorithms with optimized position calculation, the decision of adopting a simple drawing algorithm is that we want to keep the position of these components intact so that the flow from one state to the next is not broken hence breaking the mental transitions of users.

6 Miscellaneous Work

6.1 Testing Methodology

Test Driven Development (TDD) is adopted when developing the CSEC Machine. All test cases written are system tests from parsing to being evaluated. This is justifiable as too much effort is required to come up with the necessary components such as stubs to have unit and integration testing. System testing is a fair compromise in this case as the priority of the project should be on the CSEC Machine and Visualizer.

Testing is done via tracing of the machine components. However, due to time constraints, only control and stash are being traced. Tracing is done whenever a new value is pushed. Traceable control and stash stubs are used to prevent memory storage issues in production code.

The test suite is considered quite comprehensive as most edge cases are tested to ensure correctness. The test cases are categorized by features and are broken down into smaller files to prevent very large files.

6.2 Integration into Source Academy

Source Academy supports multiple languages already, e.g., JavaScript, Python, etc. Adding a new language is trivial as there is already infrastructure support.

However, there is a compatibility issue - although there are existing other languages, as these languages are largely dependent on js-slang, internal structures from js-slang are imported and used without abstraction in frontend. Note that this such an integration is valid since the need of abstraction did not arise.

With the addition of languages that do not depend on js-slang, the proper handling of these languages is to provide an abstraction that configures the shared data structures, e.g., Context, whenever users select a different language. However, as much refactoring is needed as there were too much coupling between frontend and js-slang which is out of the scope of this project, hence a less ideal adaptor approach is taken for now. For instance, the corresponding information of the java-slang Context is translated to the js-slang Context.

6.3 Integration with other Java-related Projects

Apart from the CSE Machine, java-slang houses other components, e.g., type checker, compiler and JVM. Each of these components resides in their own directory and exposes the underlying machine via an entry point function.

There is also a parser shared by the CSE Machine, type checker and compiler. A third-party library, i.e., java-parser, is used to lex and parse Java source code into a Concrete Syntax Tree (CST), and a custom visitor is written to convert the CST into an Abstract Syntax Tree (AST) with respect to the structure on consensus between the corresponding components.

Unlike the approach taken by js-slang where only one entry point function is provided and branching to the different components in done in the function, java-slang decides to provide multiple entry point functions to each component. The branching is done in frontend instead.

7 Conclusion

7.1 Limitations

Below are some limitations of the current implementation which can be readily resolved but not due to time constraint.

7.1.1 Formal Parameter Declaration and Assignment

Like fields, parameters are always initialized to their argument values. However, the declaring of formal parameters and assigning of argument values is neglected during method invocation. Such handling of formal parameters poses an inconsistency to handling of local variables and fields.

This inconsistency can be eliminated by only declaring parameters when evaluating method invocation and prepending the assignments of argument values to parameters at the start of method bodies.

7.1.2 Field Access and Typing

Representing reference with a box may not be ideal as the presence of a box which signifies memory allocation is inaccurate. The purpose of class field references is just to ease tracing of name resolution and should not interfere with the indication of memory allocation via a box. A simple fix would just be to remove the box for references.

7.1.3 Avoiding Recursion

The current underlying implementation of both qualified name/method name resolution and type inference of qualified names are recursive in nature. The presence of recursion introduces conditional decisions which is believed to be a bad heuristic for the simplicity of a notional machine.

Though they are of different extent, where EVAL_VAR recursed more than RES_TYPE. RES_TYPE only recurse once and this is a suitable amount as we want to decouple the handling of RES_TYPE and any preceding evaluations, e.g., method resolution. A similar pattern could be applied to EVAL_VAR in this case.

```
} else if (value.kind === "ExpressionName") {
  if (isQualified(value.name)) {
    const nameParts = value.name.split(".");
    for (const namePart of nameParts.slice(1)) {
      control.push(instr.resTypeContInstr(namePart, command.srcNode));
    }
    control.push(instr.resTypeInstr(node.exprNameNode(nameParts[0], command.srcNode), command.srcNode));
    return;
  }
```

RES_TYPE evaluation.

```
if (isQualified(command.symbol)) {
  const nameParts = command.symbol.split(".");
  const name = nameParts.splice(0, nameParts.length - 1).join(".");
  const identifier = nameParts[nameParts.length - 1];
  control.push(instr.resInstr(identifier, command.srcNode));
  control.push(instr.evalVarInstr(name, command.srcNode));
  return;
}
stash.push(environment.getVariable(command.symbol));
```

EVAL_VAR evaluation.

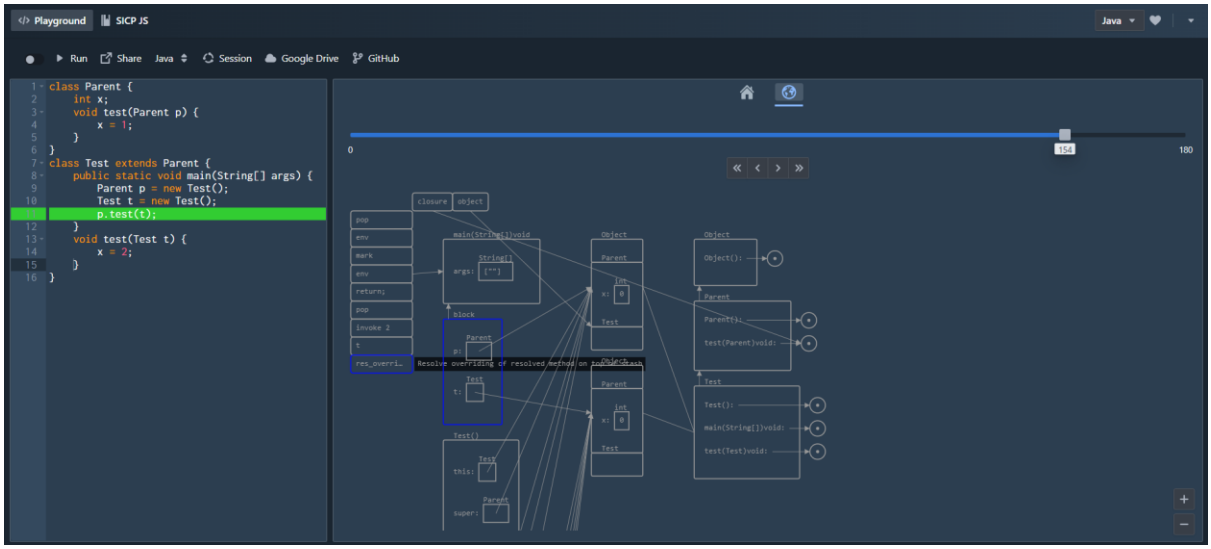
7.1.4 Drawing Algorithm

The width of control items is not dynamic and drawing algorithm could be designed to be more straightforward to advocate for simple code and to be less bug prone.

Like how sections in the environment are drawn from left to right to determine the width of the left-hand side section, such an approach can be easily extended to the control as well. However, as control items have references to things in the environment, arrows have to be drawn after all components have been laid down.

7.1.5 Method Resolution

As method overriding resolution only necessitates the method signature of the method overloading resolved method, it is sufficient to push the method signature onto stash instead of the entire method. The taking in of the entire method by the method overriding resolution step blurs the focus on only the method signature is required in method overriding resolution.



The method signature test(Parent) should be pushed onto stash instead.

7.2 Future Work

7.2.1 Next Incremental Java CSEC Machine Sublanguage Scope

The next incremental scope could include closures when inner classes are introduced. This will result in two circles for method objects where the second circle will point to the enclosing class.

7.2.2 Testing Methodology

More systematic testing could be done to improve the testing flow. Currently, there are no unit tests and integration tests, and the testing infrastructure is not on par. More test utilities, e.g., Java file readers, stubs, etc., could be written to improve testing flow.

Environment should be tested as well by tracing the values of each name binding. However, making frames traceable is not as straightforward as making stack in control and stash traceable as frames are contained inside the Environment.

7.2.3 Method Resolution

While having the this keyword as yet-another-variable works fine in method resolution, having the super keyword as yet-another-variable breaks down during method overriding resolution since method invocation using the super keyword behaves differently than other qualified method invocation. Hence, the handling of the super keyword in qualified names and method names is yet to be supported.

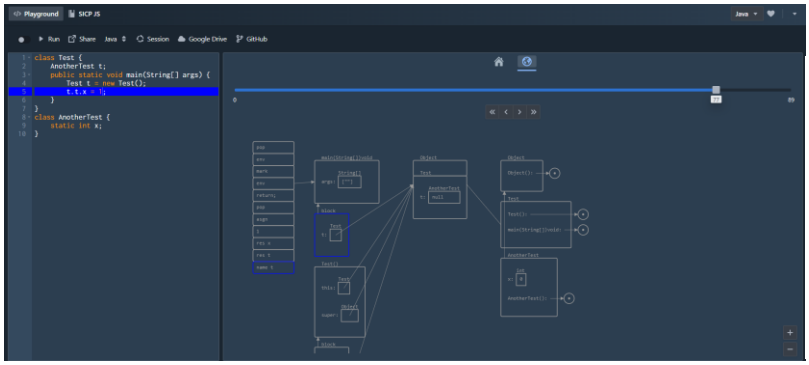
Due to time constraints, there may still be bugs lingering in evaluation of method invocation expressions, e.g., inherited methods cannot be resolved, arguments can only be of type Integer or ExpressionName.

7.2.4 Type of Qualified Names

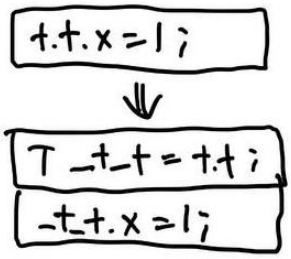
The type of qualified names is being determined dynamically, in other words, type inference is being evaluated. However, type information is supposedly static, whether type inference should be resolved dynamically or statically is up to debate. The type of qualified name and other expressions could be retrieved if the input to the CSEC machine is a typed AST, instead of a raw one. Although the type checker outputs a type annotated AST which can be used as input to the CSEC machine, due to the different scope, the parser used by the type checker is different from that of CSEC machine. Hence the typed AST is not available.

On a side note, there is a typed notional machine out there that executes type inference using tree traversal (Notional Machines, n.d.). Perhaps type inference should be handled by a separate notional machine, rather than introducing irrelevant details in the current machine which focuses on object-oriented programming.

Apart from evaluating type inference dynamically, the evaluation process can be reviewed. Currently, RES_TYPE and RES_TYPE_CONT are used to evaluate the type of expression sequentially by consuming the intermediate type. There is an alternative where transformation is done at the level of language constructs. These two approaches are the same in logic, but often higher-level syntactic explanations may be preferred, especially in introductory courses.



Lower-level machine instruction explanation.



Higher-level syntactic explanation.

However, the higher-level syntactic explanation requires temporary types and local variables which could be tricky, but nevertheless an interesting exploration.

7.2.5 Decision Points in Instruction

Some instruction now has conditional behavior. If there is too much if statements within an instruction, perhaps it is a smell/heuristic that the current instruction can be broken down in smaller parts. The presence of these underlying implicit decision points goes against the design principle of explicitness.

7.2.6 Formal Semantic Operational Semantics

There has been thought to come up with a formal specification. However, as method resolution has yet to be fully handled, it seems that we might not be ready to delve into the complexities of a vigorous mathematical specification and its practicality is debatable.

7.2.7 Obscuring

With regards to the current supported sublanguage scope, obscuring occurs when we have the same class names as variable names. In such situations, the name refers to the variable instead of the class. However, an interesting observation is that as class names exist at the global scope, class names can be looked up along the lexical environment, as per how variable names are being looked up. Such a look up suggests that classes be part of the environment, which contradicts what was mentioned in the introduction. Hence, some thoughts should be put in to formalize such a situation to distinguish between class store and environment.

7.3 The Aspiration – Java Academy

Having a glimpse into how a simple Java program with just a few lines of code unraveled into 300+ steps, Java is undeniably a beast of its own. However, with the appropriate level of abstraction, we are moving closer and closer to taming this beast and it is definitely an exciting journey to explore more concepts of the language in the future. Following the footsteps of Source Academy where a JavaScript sublanguage is examined closely, apart from general imperative language constructs, Java Academy serves as a platform to dissect the powers of the Java language and general statically typed object-oriented language constructs.

In addition, the visualization notation could potentially be formalized to be a standard, like UML diagrams.

All in all, this dissertation serves as a decent starting point to this thrilling journey ahead!

References

(n.d.). Retrieved from W3Schools: https://www.w3schools.com/java/

(n.d.). Retrieved from Programiz: https://www.programiz.com/java-programming

(n.d.). Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/java/

(n.d.). Retrieved from Javatpoint: https://www.javatpoint.com/java-tutorial

Berry, M., & Kolling, M. (2014). The state of play: a notional machine for learning programming. *ITiCSE '14: Proceedings of the 2014 conference on Innovation & technology in computer science education*, 21-26.

Boulay, B. D. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 57-73.

Boyd Anderson, M. H.-L. (2023). Community-driven Course and Tool Development for CS1. *SIGCSE TS 2023: Proceedings of the 2023 ACM SIGCSE Technical Symposium on Computer Science Education*.

CS1101S Programming Methodology. (n.d.). Retrieved from NUSMods: https://nusmods.com/courses/CS1101S/programming-methodology

CS2030S Programming Methodology II. (n.d.). Retrieved from NUSMods: https://nusmods.com/courses/CS2030S/programming-methodology-ii

Danvy, O. (2005). A Rational Deconstruction of Landin's SECD Machine. *Implementation and Application of Functional Languages*, 52-71.

Guo, P. (2021). Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia. *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology*, 1235–1251.

Harold Abelson, G. J. (2022). *Structure and Interpretation of Computer Programs JavaScript Edition.* The MIT Press.

James Gosling, B. J. (03 03, 2023). *The Java Language Specification, Java SE 20 Edition.* Retrieved from https://docs.oracle.com/javase/specs/jls/se20/html/index.html

John Clements, S. K. (2022). Towards a Notional Machine for Runtime Stacks and Scope: When Stacks Don't Stack Up. *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, 206-222.

Kaian Cai, M. H.-L.-H. (2023). Visualizing Environments of Modern Scripting Languages. *Proceedings of the 15th International Conference on Computer Supported Education, CSEDU 2023*, 146-153.

Martin Henz, L. K. (n.d.). *CS1101S: Programming Methodology, A Freshmen Module in the Department of Computer Science*. Retrieved from https://www.comp.nus.edu.sg/~cs1101s/

Notional Machines. (n.d.). Retrieved from Expression as Tree — 1. Arithmetic: https://notionalmachines.github.io/nms/ExpressionAsTree-1.html

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education, Volume 13, Issue 2*, 1-31.

Tim Lindholm, F. Y. (03 03, 2023). *The Java Virtual Machine Specification, Java SE 20 Edition.* Retrieved from https://docs.oracle.com/javase/specs/jvms/se20/html/index.html

Appendix A – Java CSEC Sublanguage Syntax

Kindly refer to the PDF attached.

Appendix B – Java CSEC Informal Structural Operational Semantics

CompilationUnit
- Check for presence of main method in at least one class, and throw error if no main method is found.
- Push main method invocation onto control.
- Push class declarations in reversed order onto control.
- Push root Object class declaration onto control.

NormalClassDeclaration
- Make method invocation expressions with simple method names qualified in instance method bodies.
- Make non-local variable non-parameter simple expression names qualified in instance method bodies.
- Append empty return statement in instance method bodies if absent.
- Make method invocation expressions with simple method names qualified in class method bodies.
- Make non-local variable non-parameter simple expression names qualified in class method bodies.
- Append empty return statement in class method bodies if absent.
- Insert default constructor in class body if not overridden.
- Prepend assignments of initializer values to instance fields in constructor bodies if alternate constructor invocation is absent.
- Make non-local variable non-parameter simple expression names qualified in constructor bodies.
- Prepend superclass constructor invocation in constructor bodies if explicit constructor invocation is absent.
- Append return statement with this keyword in constructor bodies if absent, or replace empty return statement with this keyword.
- Push ENV, encoded with current environment, onto control.
- Extend environment from global environment if class to be declared is the root Object class, else extend from superclass environment.
- Define class in current environment (global environment).
- Push instance method declarations in reversed order onto control.
- Push class method declarations in reversed order onto control.
- Push constructor declarations in reversed order onto control.
- Push class field declarations in reversed order onto control.

Block
- Push ENV, encoded with current environment, onto control.
- Push block statements in reversed order onto control.
- Extend environment from current environment.

ConstructorDeclaration
- Define constructor in current environment (class environment).

MethodDeclaration
- Define method in current environment (class environment).

FieldDeclaration
- Declare field in current environment (class environment).

- Push POP onto control.
- Push ASSIGNMENT onto control.
- Push initializer value, or default value if initializer is absent, onto control.
- Push EVAL_VAR onto control.

LocalVariableDeclarationStatement
- If initializer is present, break down local variable declaration as follows:
    - Push assignment statement of initializer value to local variable on control.
    - Push local variable declaration statement without initializer on control.
- Else:
    - Declare local variable in current environment.

ExpressionStatement
- Push POP onto control.
- Push statement expression onto control.

ReturnStatement
- Push RESET onto control.
- Push return expression onto control.

Assignment
- Push ASSIGNMENT onto control.
- Push right-hand side expression onto control.
- Push EVAL_VAR of left-hand side expression name onto control.

MethodInvocation
- Push INVOCATION, encoded with the number of arguments, onto control.
- Push argument expressions in reversed order onto control.
- Push RES_OVERRIDE onto control.
- Push target expression name onto control.
- Push RES_OVERLOAD onto control.
- Push RES_TYPE of argument expressions in reversed order onto control.
- Push RES_TYPE of target expression name onto control.

ClassInstanceCreationExpression
- Push INVOCATION, encoded with the number of arguments, onto control.
- Push argument expressions in reversed order onto control.
- Push NEW onto control.
- Push RES_CON_OVERLOAD onto control.
- Push RES_TYPE of argument expressions in reversed order onto control.
- Push RES_TYPE of class whose constructor is invoked onto control.

ExplicitConstructorInvocation
- Push POP onto control.
- Push INVOCATION onto control.
- Push arguments in reversed order onto control.
- Push NEW onto control.
- Push RES_CON_OVERLOAD onto control.
- Push RES_TYPE of arguments in reversed order onto control.
- Push RES_TYPE of class whose constructor is invoked onto control.

Literal
- Push Literal as is onto stash.

Void
- Push Void as is onto stash.

ExpressionName
- Push DEREF onto control.

- Push EVAL_VAR, encoded with the name, onto control.

BinaryExpression
- Push BINARY_OP, encoded with the binary operator, onto control.
- Push second operand expression onto control.
- Push first operand expression onto control.

POP
- Pop top element off stash.

ASSIGNMENT
- Pop first top element off stash as value.
- Pop second top element off stash as location.
- Assign value to location.
- Push value back on stash.

BINARY_OP
- Pop first top element off stash as second operand.
- Pop second top element off stash as first operand.
- Apply encoded binary operator on first and second operands.
- Push resulting value onto stash.

INVOCATION
- Push ENV, encoded with current environment, onto control.
- Push MARKER onto control.
- Pop encoded number of arguments off stash in reversed order.
- Pop method to be invoked off stash.
- Extend environment from global environment.
- Define parameters, including this and super keyword if method to be invoked is instance method or constructor, in extended environment.
- Push method or constructor body onto control.

ENV
- Restore environment to encoded environment.

RESET
- Pop next control item off control.
- Push RESET continuously if top of control is not MARKER.

EVAL_VAR
- If encoded expression name is qualified:
    - Push RES, encoded with last part of qualified name, onto control.
    - Push EVAL_VAR, encoded with front parts of qualified name, onto control.
- Else:
    - Push variable with encoded simple expression name retrieved from current environment onto stash.

RES
- Pop variable or class off stash.
- If popped stash item is a variable:
    - Search in environment whose class defines the type of popped variable and continue searching recursively in superclass environments for the first field matching the encoded name.
    - If the found field is a class field:
        - Push found field retrieved from class environment where the field is found onto stash.
    - Else the found field is an instance field:
        - Check if value of popped variable is null, and throw error if so.

        - Search recursively from within object environment whose associated class defines the type of popped variable and continue searching recursively in superclass environments for the first field matching the encoded name.
    - Else popped stash item is a class:
        - Push class field retrieved from environment of popped class onto stash.

DEREF

- Pop variable or class off stash.
- If popped stash item is a class:
    - Push the class back onto stash.
- Else popped stash item is a variable:
    - If value of popped variable is a reference to another variable:
        - Push value of referenced variable onto stash.
    - Else value of popped variable is either a literal or object:
        - Push the literal or object onto stash.

NEW

- Bookmark the current environment.
- Retrieve the class of object to be constructed and all its superclasses.
- Create the new object.
- Declare instance fields and define class fields references for each class in the corresponding environment within the object.
- Push the new objet onto stash.
- Restore the bookmark environment.

RES_TYPE

- Retrieve encoded value to be type inferred.
- If value is a literal:
    - Push the literal type onto stash.
- Else if value is an expression name:
    - If expression name is qualified:
        - Push a list of RES_TYPE_CONT, each encoded with one part of qualified name starting from the second part, onto control.
        - Push RES_TYPE, encoded with first part of qualified name, onto control.
    - Else expression name is simple:
        - Retrieve value referred by the simple name.
        - Push the value type onto satsh.
- Else value is a class:
    - Push the class as type onto stash.

RES_TYPE_CONT

- Pop type off stash.
- Retrieve class that defines the popped type.
- Search encoded field in environment of retrieved class and continue searching in superclass environments.
- If no matching field name is found, throw error.
- Push type of field found onto stash.

RES_OVERLOAD

- Pop encoded number of arguments of argument types in reversed order off stash.
- Pop type defined by class whose method is invoked off stash.
- Retrieve class of popped defining type.
- Resolve method overloading with respect to class retrieved, encoded method name, popped argument types and class store.

- Push resolved method onto stash.
- If resolved method is an instance method:
    - Modify the encoded number of arguments in most recently pushed INVOCATION on control.

RES_OVERRIDE

- Pop target expression off stash.
- Pop method overloading resolved method off stash.
- If popped resolved method is a class method:
    - No action is required.
- Else popped resolved method is an instance method:
    - Check if value of popped target is null, and throw error if so.
    - Retrieve class of popped target object.
    - Resolve method overriding with respect to class of target object and method overloading resolved method.

RES_CON_OVERLOAD

- Pop encoded number of arguments of argument types in reversed order off stash.
- Pop type defined by class whose constructor is invoked off stash.
- Retrieve class of popped defining type.
- Resolve constructor overloading with respect to class retrieved, popped class and popped argument types.
- Push resolved constructor onto stash.

Specification of Java CSEC—2024 edition

Martin Henz, Liew Xin Yi

National University of Singapore
School of Computing

April 3, 2024

Java is a programming language and computing platform first released by Sun Microsystems in 1995. It has evolved from humble beginnings to power a large share of today's digital world, by providing the reliable platform upon which many services and applications are built. New, innovative products and digital services designed for the future continue to rely on Java, as well.

1 Syntax

A Java program is a *compilation-unit*, defined using Backus-Naur Form[1] as follows:

[1]We adopt Henry Ledgard's BNF variant that he described in *A human engineered variant of BNF*, ACM SIGPLAN Notices, Volume 15 Issue 10, October 1980, Pages 57-62. In our grammars, we use **`bold`** font for keywords, *italics* for syntactic variables, ϵ for nothing, $x \mid y$ for x or y, [x] for an optional x, x... for zero or more repetitions of x, and (x) for clarifying the structure of BNF expressions.

compilation-unit ::= *class-declaration...*

class-declaration ::= **`class`** *identifier* [**`extends`** *identifier*] *class-body*

class-body ::= **`{`***class-body-declaration...***`}`**

class-body-declaration ::= *field-declaration*
| *constructor-declaration*
| *method-declaration*

field-declaration ::= *field-modifier... type identifier* [**`=`** *expression*] **`;`**

field-modifier ::= **`static`**

method-declaration ::= *method-modifier... method-header block*

method-modifier ::= **`static`**

method-header ::= *result-type method-declarator*

result-type ::= *type* | **`void`**

method-declarator ::= *identifier* **`(`***formal-parameter...***`)`**

formal-parameter ::= *type identifer*

constructor-declaration ::= *identifier* **`(`***formal-parameter...***`)`** *block*

type ::= *primitive-type* | *reference-type*

primitive-type ::= **`int`**

reference-type ::= *identifier* | **`String[]`**

block ::= **`{`***block-statement...***`}`**

block-statement ::= *local-variable-declaration-statement*
| *expression-statement*
| *return-statement*
| *explicit-constructor-invocation*

local-variable-declaration-statement ::= *type identifier* [**`=`** *expression*] **`;`**

expression-statement ::= *expression* **`;`**

return-statement ::= **`return`** [*expression*] **`;`**

explicit-constructor-invocation ::= *primary* **`(`***expression...***`) ;`**

assignment ::= *left-hand-side* **`=`** *expression*

left-hand-side ::= *expression-name* | *field-access*

expression ::= *binary-expression*
| *assignment*
| *expression-name*
| *literal*
| *primary*
| *class-instance-creation-expression*
| *field-access*
| *method-invocation*

binary-expression ::= *expression binary-operator expression*

binary-operator ::= **`+`** | **`-`** | **`*`** | **`/`** | **`%`**

primary ::= **`this`** | **`super`**

class-instance-creation-expression	::=	**`new`** *identifier* **`(`***expression...***`)`**
field-access	::=	**`this.`***identifier*
method-invocation	::=	*expression-name* **`(`***expression...***`)`**
	|	**`this.`***identifier* **`(`***expression...***`)`**
expression-name	::=	*identifier* | *expression-name* **`.`** *identifier*
identifier	::=	*string*
literal	::=	*number*
	|	**`[""]`**
	|	**`null`**

Restrictions

- Explicit constructor invocations are only allowed in constructor bodies.
- Multiplicative binary operators, i.e., `*`, `/`, `%` have higher precedence than additive binary operators, i.e., `+`, `-`. Also, multiplicative and additive binary operators are left associative.
- Only assignments and method invocations are allowed as expression in expression statements.

Identifiers

Identifers start with a Java letter[2] and contain only Java letter-or-digit[3]. Reserved keywords[4] are not allowed as identifiers.

Numbers

Only decimal integers are supported.

Notes

- **`String[]`** and **`[""]`** are included to accommodate **`main`** method declaration and invocation.

[2] By *Java letter* we mean character for which the method **`Character.isJavaIdentifierStart(int)`** returns **`true`**.

[3] By *Java letter-or-digit* we mean characters for which the method **`Character.isJavaIdentifierPart(int)`** returns **`true`**.

[4] By *restricted word* we mean any of: **`abstract`**, **`assert`**, **`boolean`**, **`break`**, **`byte`**, **`case`**, **`catch`**, **`char`**, **`class`**, **`const`**, **`continue`**, **`default`**, **`do`**, **`double`**, **`else`**, **`enum`**, **`extends`**, **`final`**, **`finally`**, **`float`**, **`for`**, **`if`**, **`goto`**, **`implements`**, **`import`**, **`instanceof`**, **`int`**, **`interface`**, **`long`**, **`native`**, **`new`**, **`package`**, **`private`**, **`protected`**, **`public`**, **`return`**, **`short`**, **`static`**, **`strictfp`**, **`super`**, **`switch`**, **`synchronized`**, **`this`**, **`throw`**, **`throws`**, **`transient`**, **`try`**, **`void`**, **`volative`**, **`while`**, **`true`**, **`false`**, **`null`**, **`yield`**, _(underscore).