Physical Java Memory Models: A Notional Machine

Colleen M. Lewis
Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, IL, USA
ColleenL@illinois.edu

ABSTRACT

I have created a notional machine that uses physical objects to help students understand references and objects in Java programs. These physical models may help students understand the abstract representation of Java programs - much like blocks or other physical items help young children reason about abstract things like quantity, addition, and subtraction. This article and accompanying videos may be of interest to Java teachers and to researchers interested in evaluating or comparing the effectiveness of notional machines.

CCS CONCEPTS

• **Social and professional topics → Computing education**; **Computer science education**.

KEYWORDS

notional machine; Java; objects; references; APCSA

ACM Reference Format:
Colleen M. Lewis. 2021. Physical Java Memory Models: A Notional Machine. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432477

1 INTRODUCTION

Inspired by work in mathematics that uses physical objects to help students understand abstract concepts [2, 21, 22], I designed a Java memory model that includes physical objects. I have iteratively refined these physical models within 16 offerings of a course that introduces Java to post-secondary students who have taken a breadth-first CS1 course using Python. The primary contribution of this article is to offer an example of a notional machine, i.e., "a pedagogic device to assist the understanding of some aspect of programs or programming." [8, p. 502]. This paper attempts to document my pedagogical content knowledge (PCK) [16, 17] related to teaching references and objects in Java. Java teachers may wish to adopt or adapt the notional machine within their teaching. CS education researchers need examples of notional machines to be able to investigate the relative effectiveness of various notional machines.

Section 2 provides the learning goals of my notional machine, which spans multiple weeks in my course. Section 3 describes an essential element of a notional machine, the mapping of relevant concepts in the programming language to their instantiation in the notional machine. Section 4 walks through 11 examples that illustrate the learning trajectory for this notional machine. Section 4 also includes my hypotheses about the benefit of the physicality of the model and my observations about students' understanding. Section 5 includes practical considerations and Section 6 identifies connections to previous research. Section 7 summarizes my hypothesis about the primary benefit of the notional machine and learning trajectory. Videos are available for each example [9].

SIGCSE '21, March 13–20, 2021, Virtual Event, USA.

ACM ISBN 978-1-4503-8062-1/21/03.
https://doi.org/10.1145/3408877.3432477

2 LEARNING GOALS

These correspond with the 11 examples in Sections 4.1 to 4.11.

(1) Two `int` variables are independent and do not become linked even if one gets set to the value of the other.
(2) Calling a method with an argument is another way to assign a value to a variable.
(3) A variable can hold a reference to an `int` array.
(4) If a variable can hold a reference to an `int` array and we do not set the value of that variable, we have a null reference.
(5) Two variables can reference the same `int` array and both of them can modify the values in the array.
(6) An array passed to a method can be modified within that method.
(7) Calling a constructor makes an object.
(8) If a variable can hold a reference to an object and we do not set the value of that variable, we have a null reference.
(9) Two variables can reference the same object and both of them can modify that object.
(10) Calling a method on an object provides access to a variable named `this` within the method and the variable `this` references the same object that the method was called on.
(11) An object contains the values of each of the instance variables in the class.

3 NOTIONAL MACHINE MAPPING

An essential element of a notional machine is a mapping between a concept in the programming language and the instantiation of that concept in the notional machine. Each mapping in the list below identifies the section in which that mapping is introduced. Footnotes describe the construction of the physical model. These mappings are cumulative for Examples 1 through 10. Example 10 and 11 model the same Java program and Example 11 introduces a new mapping for objects and instance variables.

- **Stack frame**: A rectangle with the name of the method on the left-hand side (Example 1).
- **Local variables**: Written within a stack-frame rectangle; the variable name goes to the left of the value (Example 1).

- **`int` variables**: A pocket that holds a piece of paper with a number on it (Example 1)[1].
- **`String[] args`**: Not represented (Example 1)[2].
- **Class/`static` variables**: Not represented (Example 1) [3].
- **Method Call**: A new stack frame drawn above the stack frame of the calling method (Example 2).
- **One-dimensional `int` array**: A sequence of `int` variables (Example 3).
- **`int[]` variable**: A pocket (Example 3)[4].
- **Reference to an `int` array**: An remote control with a blue ribbon and magnetic clip showing the referent (Example 3)[5].
- **`null`**: An empty pocket with a white ribbon in the back making an X; the X is not visible if there is a remote control (i.e., reference) in the pocket (Example 4).
- **Array access**: Following a ribbon from a remote control to the corresponding array and then counting to find the correct index within the array (Example 5).
- **Object**: A toy with some resemblance to the type of the object (Example 7).
- **Reference to an object**: Identical to a reference to an `int` array (Example 7).
- **Calling a method on an object**: Following a ribbon from a remote control to the object it is keeping track of (Example 9).
- **Modifying an instance variable**: Following a ribbon from a remote control to the object it is keeping track of and saying, with intentional vagueness, that we could update the object (Example 10).
- **Object**: A rectangle with the name of the class written at the top and underlined (Example 11).
- **Instance Variables**: Written within the rectangle representing the object (Example 11).

4 LEARNING TRAJECTORY

Each of the 11 examples include a Java program written in the BlueJ IDE (https://www.bluej.org/), a picture of a physical memory model, and a learning goal (in the figure caption). Subsections begin with what I try to draw my students' attention to. Later paragraph(s) describe my hypotheses about the value of the physicality.

4.1 Example 1: `int` Variables

Using Example 1, I try to draw students' attention to the fact that: *(i)* When we change the value of `x`, it does not change the value of `y`. *(ii)* The variable `y` "gets" a *copy* of the *value* of the variable `x`.

```
public class Example1{
    public static void main(String[] args){
        int x = 44;
        int y = x;
        x = 15;
    }
}
```

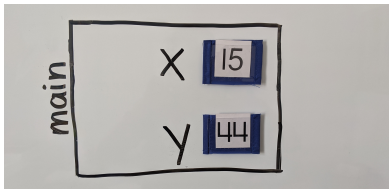

Figure 1: Learning Goal - Two `int` variables are independent and do not become linked even if one gets set to the value of the other.

Introducing variables of type `int` does not seem to require or noticeably benefit from a physical representation. I quickly transition to simply writing a number in a small box next to the variable name, but use the physical representation when introducing arrays (see Figure 3) and when comparing the primitive types `int`, `double`, and `boolean`[6].

4.2 Example 2: Method Calls have Scope

```
public class Example2{
    public static void methodA(int x){
        x = 15;
    }
    public static void main(String[] args){
        int x = 44;
        methodA(x);
    }
}
```

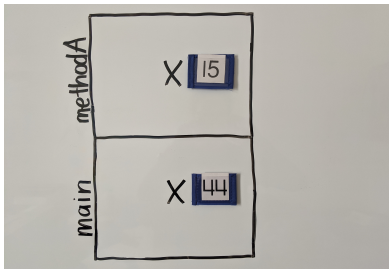

Figure 2: Learning Goal - Calling a method with an argument is another way to assign a value to a variable.

Using Example 2, I try to draw students' attention to the fact that: *(i)* Changing the value of `x` in `methodA` does not change the value

[1] I make the `int` variables by cutting up a 10x10 display designed for displaying the numbers 1 to 100 in a classroom. I glue magnets to the back of all of my physical objects or hang them using magnetic clips.

[2] I usually do not represent the variable `args` in the stack frame. In most cases, the value of the variable will be a zero-length `String` array and I do not have a good way to visualize that. When I do represent it, I explain that we represent it as if it were `null` even though it is a zero-length `String` array.

[3] I have not yet figured out a good way to represent class/`static` variables without making the representations too complicated. I tell students that since `static` variables are shared by all objects of the class (i.e., there is only one), it is not as necessary to have a formal way to represent it.

[4] I make the variable pocket that can hold a reference from a 6 by 6 grid of pockets designed to hold a classroom set of calculators or cell phones.

[5] The remote control in the photos is an iClicker, a classroom response device, but I have also used cardboard jewelry boxes because they have the size and shape of a TV remote control.

[6] No examples are provided showing variables of type `double` and `boolean` because it does not relate closely to the instructional sequence. However, I think that showing all three can be helpful for reinforcing the importance of type. A variable of type `double` is similar to those of type `int`, but the pocket holds a piece of paper that is the size of an index card and I write a decimal value on the card. A variable of type `boolean` is represented as a switch, which has states labeled "on" and "off." I find the `boolean` variables particularly helpful because students have difficulty reasoning about booleans as state.

of x in the main method. *(ii)* We can have two variables named x because each one is in a different method and therefore has a different scope. *(iii)* Similar to Example 1, two int variables do not become linked if one gets set to the value of the other. *(iv)* Parameters become local variables within a method.

The example in Figure 2 does not seem to require or noticeably benefit from a physical representation. The particular example is an important comparison case for Example 4.6 where physicality does seem valuable.

4.3 Example 3: Reference to an Array

```
public class Example3{
    public static void main(String[] args){
        int[] arrA = {42, -10, 29};
        int x = 44;
    }
}
```

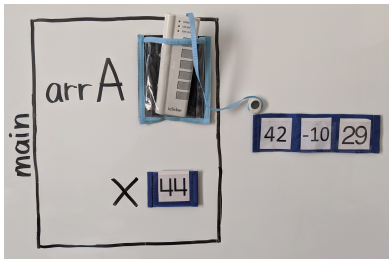

Figure 3: Learning Goal - A variable can hold a reference to an int array.

Using Example 3, I try to draw students' attention to the fact that: *(i)* An int array of some sizes would not fit neatly in our stack frame. *(ii)* The stack frame includes a variable that references an array. *(iii)* An int array is just a sequence of int variables.

In illustrating that we want a reference to keep track of the int array, I show arrays of different sizes and two-dimensional int arrays. The larger of these do not fit neatly within the stack frame. Students seem to accept the idea that this motivates the use of a reference, which I believe is a benefit of the physical representation.

The *values* at each index are written on paper in the pocket and, when relevant, I write the *indices* below those elements on the chalkboard. I think this helps avoid the confusion between the value of an array element and the index of that array element.

Additionally, the physical representation illustrates that an int array is a sequence of int variables, but I do not have any evidence that students benefit from or appreciate this consistency.

4.4 Example 4: Null Reference

Using Example 4, I try to draw students' attention to the fact that: *(i)* arrB is a variable that could reference an int[] (i.e., an int array), but has not been set. *(ii)* Since the variable arrB does not contain a remote control, we can see the X on the back of the pocket, which we use to represent null, and X is a common way to represent null. *(iii)* If we try to use the variable arrB, we get a NullPointerException because we have an empty pocket, but try to use a remote control in it.

```
public class Example4{
    public static void main(String[] args){
        int[] arrA = {42, -10, 29};
        int[] arrB;
    }
}
```

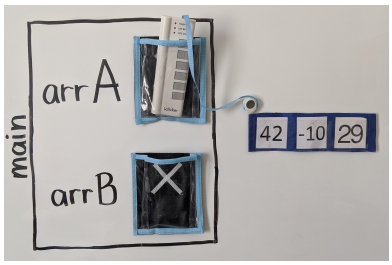

Figure 4: Learning Goal - If a variable can hold a reference to an int array and we do not set the value of that variable, we have a null reference.

Before I developed the physical representation, students had a *lot* of questions about null. Similar to students' difficulty with empty words and sentences in Scheme [3], their questions were often "how is it both *nothing* and *something*?" It goes much better now that I describe a null reference as an empty pocket. Later, I draw null as a box with an X through it, but I bring an empty pocket to every class in case null comes up because I think it helps.

4.5 Example 5: Aliasing Arrays

```
public class Example5{
    public static void main(String[] args){
        int[] arrA = {42, -10, 29};
        int[] arrB = arrA;
        arrA[2] = 99;
    }
}
```

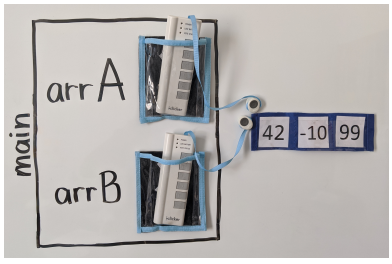

Figure 5: Learning Goal - Two variables can reference the same int array and both of them can modify the values in the array.

Using Example 5, I try to draw students' attention to the fact that: *(i)* The variable arrB "gets" a *copy* of the *value* of the variable arrA; the *value* of the variable is a *reference*. *(ii)* The variable arrB was set in exactly the same way as we saw in Section 4.1: "The variable y "gets" a *copy* of the *value* of the variable x." *(iii)* The line int[] arrB = arrA; does not create a copy of the array. *(iv)* Both arrA and arrB can modify the values of the array. *(v)* The variables arrA

and arrB do not become linked and either one could be changed without impacting the other.

The physicality helps to show that we do not make a copy of the array and that both arrA and arrB reference the same array and are not linked in any other way. My students do not seem to find aliasing confusing in the context of arrays, but do in the context of objects (see Section 4.9).

4.6 Example 6: Method Calls

```
public class Example6{
    public static void methodA(int[] inputArr){
        inputArr[2] = 99;
    }
    public static void main(String[] args){
        int[] arrA = {42, -10, 29};
        methodA(arrA);
    }
}
```

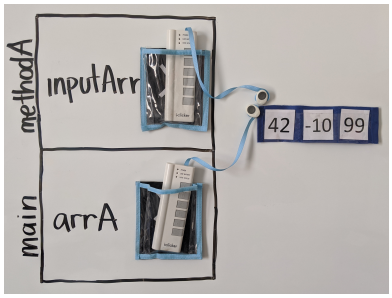

Figure 6: Learning Goal - An array passed to a method can be modified within that method.

Using Example 6, I try to draw students' attention to the fact that: *(i)* The variable inputArr "gets" a *copy* of the *value* of the variable arrA; the *value* of the variable is a *reference*. *(ii)* In methodA we can modify the array using the variable inputArr. *(iii)* In methodA we cannot access the variable arrA.

The physicality does not seem particularly helpful here but builds towards the introduction of the variable this as described in Section 4.10.

Although my students seem to find this example unproblematic, on a homework assignment they seem to have trouble applying the idea that an array passed to a method can be modified within that method. In their homework assignment they need to modify two arrays and often express that it would be impossible to do that within a method because they can only return one array from the method. Reminding them of Example 6 seems to resolve their confusion. However, I have more work to do to understand what makes the idea hard to apply.

4.7 Example 7: Constructors

Using Example 7, I try to draw students' attention to the fact that: *(i)* A constructor is a method with the same name as the name of the class, and we call a constructor using the keyword new. *(ii)* We called the constructor twice and so have two separate objects. *(iii)* If each of the Dog objects had an age, they would each have their own copy of that property because they are separate objects. *(iv)*

```
public class Example7{
    public static void main(String[] args){
        Dog fido = new Dog();
        Dog spot = new Dog();
    }
}
```

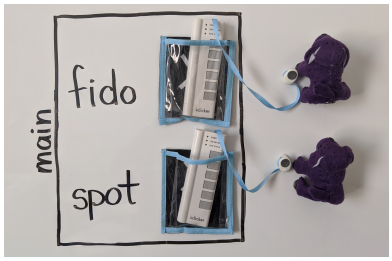

Figure 7: Learning Goal - Calling a constructor makes an object.

The variables fido and spot are variables that can hold a reference to a Dog.

Representing an object as a toy allows me to ignore instance variables and therefore ignore any details about the body of a constructor call. I can still introduce the idea that each of the objects would have their own copy of any properties. When I later introduce the difference between == (i.e., "equal equal") and .equals (i.e., "dot equals"), I reuse Examples 7 and 9.

4.8 Example 8: Null References

```
public class Example8{
    public static void main(String[] args){
        Dog fido = new Dog();
        Dog spot;
    }
}
```

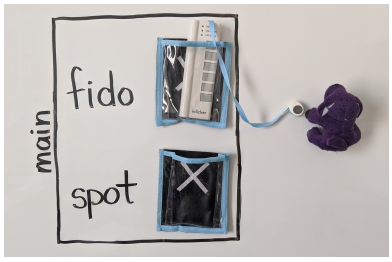

Figure 8: Learning Goal - If a variable can hold a reference to an object and we do not set the value of that variable, we have a null reference.

Using Example 8, I try to draw students' attention to the fact that: *(i)* spot is a variable that could reference a Dog object, but has not been set. *(ii)* The variable spot is not a Dog object. *(iii)* If we try to use the variable spot, we get a NullPointerException because we have an empty pocket, but try to use a remote control in it.

The physical pocket with an X in the back representing null seems particularly helpful here and in Section 4.4. The example is particularly helpful in distinguishing between the type of a variable and the actual object.

4.9 Example 9: Aliasing Objects

```
public class Example9{
    public static void main(String[] args){
        Dog fido = new Dog();
        Dog spot = fido;
    }
}
```

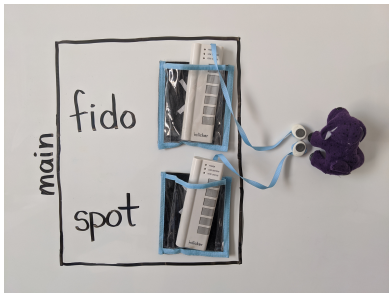

Figure 9: Learning Goal - Two variables can reference the same object and both of them can modify that object.

Using Example 9, I try to draw students' attention to the fact that: *(i)* The variable `spot` "gets" a *copy* of the *value* of the variable `fido`; the *value* of the variable is a *reference*. *(ii)* The line `Dog spot = fido;` does not create a copy of the original `Dog` object. *(iii)* If a `Dog` object has a property like an age, either variable could be used to modify the age of the `Dog`.

Students often assume that a statement like `Dog spot = fido;` will create a copy of the original `Dog`. That would be a reasonable thing for a programming language designer to decide, but it is not what Java does. The example and the physical representation are designed to dispel that misconception and seem effective.

More importantly, the physical model of Example 9 generates more and better questions than I had before. Students typically make the *correct* inference from the model, but express disbelief. They often believe that they are pointing out a mistake that I have made. They will explain that the way I have represented it would mean that both `fido` or `spot` could modify the object. They can. Students' reactions to the example make me certain that it is addressing a key conceptual difficulty even if I do not fully understand what makes it difficult. I am further puzzled that my students do not express the same confusion when I introduce aliasing with references to `int` arrays as is Figure 5.

The physical representation also allows me to talk about a `Dog` object as having properties, like an age, without discussing instance variables in detail.

4.10 Example 10: The Variable `this`

Using Example 10, I try to draw students' attention to the fact that: *(i)* We did not explicitly create the variable named `this`, it was provided by Java. *(ii)* The variable `this` and `fido` reference the same object. *(iii)* In `getOlder` we cannot access the variable `fido`.

Being able to separate the introduction of the variable `this` from instance variables seems to be helpful. I start introducing the idea that a `Dog` object could have an age in previous examples and only

```
public class Dog{
    private int myAge;
    public void getOlder(){
        this.myAge++;
    }
    public static void main(String[] args){
        Dog fido = new Dog();
        fido.getOlder();
    }
}
```

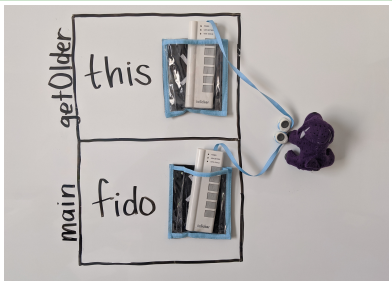

Figure 10: Learning Goal - Calling a method on an object provides access to a variable named `this` within the method and the variable `this` references the same object that the method was called on.

here start showing the code for an instance variable `myAge`. However, I do not start representing instance variables in the memory model until the next example.

4.11 Example 11: Instance Variables

```
public class Dog{
    private int myAge;
    public void getOlder(){
        this.myAge++;
    }
    public static void main(String[] args){
        Dog fido = new Dog();
        fido.getOlder();
    }
}
```

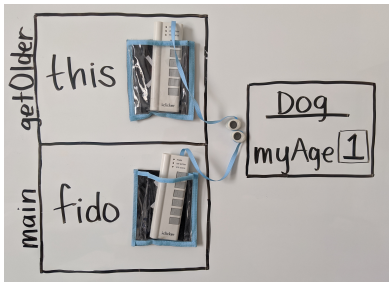

Figure 11: Learning Goal - An object contains the values of each of the instance variables in the class.

Figure 11 includes the same code as Figure 10. The important points in Section 4.11 are equally relevant here even though there is a different representation. Additionally, I try to draw students' attention to the fact that: *(i)* Each object has its own copy of all

instance variables. *(ii)* Stack frames and objects are drawn in similar ways, which is unfortunate because they are conceptually distinct.

Here I transition almost entirely away from the use of the physical models. To reinforce the fact that each object gets its own copy of each instance variable, I would draw an object with more than one instance variable and redraw the example in Section 4.7, which involves two `Dog` objects.

5 PRACTICAL LESSONS LEARNED

The most important thing that my students have coached me on is that I should not trace more than a few lines of code. My instinct was to trace through "interesting" and "complicated" code to show how the physical models could help us keep track of what was happening. My students reported that it was too confusing to follow. Now, if I have a complicated example, I will set it up before class to show the state of memory immediately before a crucial step. However, with my drive to smaller and smaller examples, like you see here, students will very reasonably ask "what is the point of the code?" My answer, "there is no point," is deeply unsatisfying to me and my students, but I think that the complexity of these ideas justify introduction with short, uninteresting programs.

As I began developing the notional machine, I received student feedback that it was infantilizing. I have not received those critiques in the last few years, and I primarily attribute that to my better ability to explain the goals of the physical models and their increased effectiveness. As my notional machine has evolved, stuffed animals are also a smaller percentage of the physical models. In the early years, I was only using stuffed animals (to represent objects) and boxes (to represent references).

6 CONNECTIONS TO PREVIOUS RESEARCH

6.1 Notional Machines

Early research related to the programming language Logo created strategies for explaining the semantics of the programming language [6]. The article "The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices" [7] described these strategies as a "notional machine." du Boulay introduced the term in the 1980s [5, 7] and more recent work has popularized the term [15, 19]. Recent work argues for the documentation of notional machines currently in use [8]. The use of notional machines seems to be common in CS, but there is a gap in our understanding of their effectiveness [19]. While notional machines are embedded in instructional materials (e.g., textbooks, assignments, and lectures), to test the effectiveness of notional machines we need clear descriptions; this paper attempts to contribute that.

6.2 Concrete Representational Abstract

Abstraction is frequently mentioned as a core skill developed when learning programming [1, 4, 10, 20], but CS education rarely draws on education research focused on helping students build their understanding of abstraction using physical objects [2, 21, 22].

Richard Mayer, an early researcher in CS education, argued for the use of physical objects to aid conceptual understanding in CS [11]. Within mathematics education, the use of physical objects is common, and these objects are referred to as "manipulatives" because the student can directly interact with them. In a meta-analysis of studies of mathematics instruction, there were small to moderate effect sizes "in favor of the use of manipulatives when compared with instruction that only used abstract math symbols." [2, p. 380]. Another meta-analysis found that manipulatives and drawings were an effective practice for improving students' performance on computation tasks and word problems [12].

A specific instructional sequence has developed around the use of manipulatives, which is known as concrete-to-representational-to-abstract (CRA) [22]. For example, teaching addition could involve physical blocks (i.e., a concrete form), pictures of blocks (i.e., a representational form), and eventually numbers (i.e., an abstract form). Witzel, Riccomini, and Schneider [22] argue that the practice helps students connect the concrete and abstract forms [22], which is necessary for understanding and working with the abstract forms. A meta-analysis focused on instructional strategies to support low-performing students found that the CRA strategy had the largest effect sizes [2]. Given the pervasive use of abstract forms in CS (e.g., programming language syntax), the use of concrete manipulatives as a bridge to help students understand these abstract forms may be a promising direction for CS education.

7 CONCLUSION

The primary goal of my notional machine is to help students understand references and objects in Java programs. The set of 11 examples captures a learning trajectory, but each example contributes in different ways as described below.

To help students understand references, Example 3 motivates the need for a reference and Example 4 helps students understand null. Additionally, Examples 1 and 5 illustrate that assignment of variables that hold primitives and references work identically in that they receive a *copy* of the value they are assigned. Perhaps most impactful, Example 9 shows that two variables can refer to the same object. My students do not seem to be confused by similar aliasing with `int` arrays as shown in Example 5, but Example 9 provokes a lot of questions. Students typically make the *correct* inference from the model, but often believe their inference is inconsistent with Java semantics (i.e., that the model is wrong).

To help students understand objects, Example 7 allows me to introduce the idea of making a new object without discussing constructors. Then, Example 10 allows me to introduce the variable `this` without discussing instance variables. And Example 10 built upon Examples 2 and 6, which illustrated method calls. Finally, Example 11 allows me to introduce instance variables.

Consistent with previous research [13, 14], I have had difficulty teaching students the basics of references and objects in Java. Using this notional machine seems to increase the quality and quantity of questions my students ask as they are learning about Java references and objects.

ACKNOWLEDGMENTS

I owe the foundation of my approach to the memory diagrams drawn by Jonathan Shewchuk and the metaphor of a reference as a remote control introduced in Head First Java [18]. I am beyond thankful for the hundreds of students who helped me refine these memory models. This material is based upon work supported by the National Science Foundation (1339404, 1758455, & 1821136).

REFERENCES

[1] Jens Bennedssen and Michael E Caspersen. 2008. Abstraction ability as an indicator of success for learning computing science?. In *Proceedings of the Fourth international Workshop on Computing Education Research*. ACM, Association for Computing Machinery, New York, NY, USA, 15–26.

[2] Kira J Carbonneau, Scott C Marley, and James P Selig. 2013. A meta-analysis of the efficacy of teaching mathematics with concrete manipulatives. *Journal of Educational Psychology* 105, 2 (2013), 380.

[3] Michael Clancy. 2004. Misconceptions and attitudes that interfere with learning to program. In *Computer Science Education Research*, Sally Fincer and Marian Petre (Eds.). Oxford University Press, Oxford, Chapter 1, 85–100.

[4] Quintin Cutts, Sarah Esper, Marlena Fecho, Stephen R. Foster, and Beth Simon. 2012. The Abstraction Transition Taxonomy: Developing Desired Learning Outcomes through the Lens of Situated Cognition. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12)*. Association for Computing Machinery, New York, NY, USA, 63–70. https://doi.org/10.1145/2361276.2361290

[5] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (Feb. 1986), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

[6] B. du Boulay and T. O'Shea. 1976. *How to work the LOGO Machine: a primer for EL-OGO*. University of Edinburgh, Department of Artificial Intelligence, Edinburgh, Scotland. Type: Book.

[7] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies* 14, 3 (1981), 237–249. http://www.sciencedirect.com/science/article/B6WGS-4T7YSPP-2/2/ea0352066f4fcc9d43d666d0c872090b Type: Journal Article.

[8] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Capturing and Characterising Notional Machines. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 502–503. https://doi.org/10.1145/3341525.3394988

[9] Lewis, Colleen M. 2020. Physical Java Memory Models: A Notional Machine. https://www.youtube.com/playlist?list=PLHqz-wcqDQIE6nNE58CaEoHJKhSD_4J4S. Accessed: 2020-11-26.

[10] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further Evidence of a Relationship Between Explaining, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09)*. ACM, New York, NY, USA, 161–165. https://doi.org/10.1145/1562877.1562930

[11] Richard E. Mayer. 1981. The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13, 1 (March 1981), 121–141. https://doi.org/10.1145/356835.356841

[12] Susan Peterson Miller, Frances M Butler, and Lee Kit-hung. 1998. Validated practices for teaching mathematcis to students with learning disabilities: A review of literature. *Focus on Exceptional Children* 31, 1 (1998), 1.

[13] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct. 2017), 24 pages. https://doi.org/10.1145/3077618

[14] Noa Ragonis and Mordechai Ben-Ari. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education* 15, 3 (2005), 203–221. https://doi.org/10.1080/08993400500224310

[15] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education* 13, 2 (2003), 137–172. Publisher: Taylor & Francis.

[16] Lee Shulman. 1987. Knowledge and Teaching: Foundations of the New Reform. *Harvard Educational Review* 57, 1 (April 1987), 1–23. http://hepg.metapress.com/content/J463W79R56455411

[17] L. S. Shulman. 1986. Those Who Understand: Knowledge Growth in Teaching. *Educational Researcher* 15, 2 (Feb. 1986), 4–14. https://doi.org/10.3102/0013189X015002004

[18] Kathy Sierra and Bert Bates. 2005. *Head First Java: A Brain-Friendly Guide.* " O'Reilly Media, Inc.", Sebastapool, Canada.

[19] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 2 (July 2013), 1–31. https://doi.org/10.1145/2483710.2483713 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[20] Leigh Ann Sudol-DeLyser. 2015. Expression of Abstraction: Self Explanation in Code Production. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 272–277. https://doi.org/10.1145/2676723.2677222

[21] Bradley S Witzel, Paul J Riccomini, Karen M Fries, and Gibbs Y Kanyongo. 2014. A Meta-Analysis of Algebra Interventions for Learners with Disabilities and Struggling Learners Elizabeth M. Hughes Duquesne University Hughese1@ duq. edu. *The Journal of the International Association of Special Education* 15, 1 (2014), 36.

[22] Bradley S Witzel, Paul J Riccomini, and Elke Schneider. 2008. Implementing CRA with secondary students with learning disabilities in mathematics. *Intervention in School and Clinic* 43, 5 (2008), 270–276.