

# Hybrid and Non-Hybrid Block-Based Programming Languages in an Introductory College Computer-Science Course

Journal of Educational Computing  
Research  
0(0) 1–27

© The Author(s) 2021


Article reuse guidelines:

sagepub.com/journals-permissions

DOI: 10.1177/0735633120985108

journals.sagepub.com/home/jec



Wen-Chin Hsu<sup>1</sup>  and  
Julie Gainsburg<sup>2</sup>

## Abstract

Block-based programming languages (BBLs) have been proposed as a way to prepare students for learning to program in more sophisticated, text-based languages, such as Java. Hybrid BBLs add the ability to view and edit the block commands in auto-generated, text-based code. We compared the use of a non-hybrid BBL (Scratch), a hybrid BBL (Pencil Code), and no BBL across three sections of an introductory CS course that taught Java programming, to determine whether either type of BBL offered cognitive or affective advantages for learning Java. Students in the BBL groups were surveyed about their perceptions of each BBL in terms of ease of use and helpfulness in learning Java, and all three groups were compared on their performance in Java programming. The results showed that, in this introductory CS course, neither type of BBL offered an advantage in preparing students for learning

---

<sup>1</sup>Department of Computer Science, College of Engineering and Computer Science, California State University, Northridge, United States

<sup>2</sup>Department of Secondary Education, Michael D. Eisner College of Education, California State University, Northridge, United States

## Corresponding Author:

Wen-Chin Hsu, Department of Computer Science, College of Engineering and Computer Science, California State University, 18111 Nordhoff St., Northridge, CA 91330, United States.

Email: [hsu.wen-chin@csun.edu](mailto:hsu.wen-chin@csun.edu)

Java. These results held regardless of the students' level of Java knowledge prior to the course.

**Keywords**

block-based visual programming, textual languages, dual-modality programming environments, hybrid, and CS0

A significant issue for computer science education researchers is the high failure rate in introductory college programming courses (Vihavainen et al., 2014). Studies by Watson and Li (2014) and Bennedsen and Caspersen (2007) estimate the worldwide failure rate at about 30%. Failure or poor performance in an introductory programming course, such as “CS1,” the standard first-year computer-science course, can discourage students and cause them to abandon the computer-science major (Bergin & Reilly, 2005). This situation may be contributing to the observed decline in computer science majors (Watson & Li, 2014).

One approach to solving this problem has been establishing a “Computer Science Orientation” (CS0) course—a preparatory (but usually non-prerequisite) course for CS1 (Cook, 1997; Vihavainen et al., 2014). Another approach has been to develop new programming languages that are specifically designed for teaching programming—transitional tools to facilitate the learning of more sophisticated, industry-standard languages such as Java. These new languages, which include Alice, Scratch, and MIT App Inventor, share two main pedagogical features. First, their “block-based” structure greatly reduces the amount of syntax (coding grammar) that students must learn (Resnick et al., 2009), enabling them to more quickly understand broader programming concepts (Bau et al., 2017; Kaucic & Asic, 2011). Second, these block-based languages (BBLs) make it easy to program interactive stories, games, and animations, which can be fun and motivating for students and can simplify error-detection and debugging (Moors et al., 2018). In other words, BBLs are theorized to have both affective and cognitive benefits for learners. BBLs were originally developed for children, but they have increasingly been incorporated into college-level CS0 and CS1 courses (Kaucic & Asic, 2011; Resnick et al., 2009; Rizvi & Humphries, 2012).

More recently, hybrid languages have emerged in K-12 computer-science education. Hybrid languages are block-based, but they allow students to “toggle” to a code view, where they can read and, in some cases, edit a translation of the block command into a more traditional, text-based language. Hybrid BBLs ostensibly facilitate the transition from non-hybrid BBLs to Java or other traditional, text-based languages (Bau et al., 2017).

Because hybrid BBLs are relatively new, their use, especially in college programming courses, has been little studied. This is unfortunate, given the hunger of CS educators for ways to attract and retain CS majors. Presumably, a tool that helped students learn the language used in CS1 and later courses would mitigate failure and dropout rates. In this article, we report on an investigation of student perceptions of a hybrid and a non-hybrid BBL in an introductory, college computer science (CS0) course and the comparative impact of these tools on learning Java.

## Literature Review

### *Purposes for Teaching Programming*

Computer-science educators commonly grapple with the question of what programming language to use in the classroom (Kolling, 1999; Pane et al., 2001; Pears et al., 2007). This decision should take into account the course's purpose for teaching programming (Pears et al., 2007). One main purpose for teaching programming is to prepare programmers, either for programming jobs or for future courses that require programming (Bau et al., 2017). For this purpose, the choice of language should reflect the profession, ultimately training students to code in Java, C, C++, or another industry standard. An alternative purpose for teaching programming is to developing general problem-solving and thinking skills that are presumed to be useful and usable in non-programming situations (Bau et al., 2017; Klahr & Carver, 1988; Pears et al., 2007). The term computational thinking has been used to encompass a range of competencies that are related to programming but more broadly applicable, including understanding abstractions and pattern generalizations; using symbol systems and representations; iterative, recursive, and parallel thinking; and conditional logic (Grover & Pea, 2013; Lye & Koh, 2014). For this purpose, the choice of programming language is irrelevant as an outcome, because learning that language is not the primary goal. An easier-to-learn language would be a sensible choice in this case, on the theory that the general cognitive skills can be gained from programming experience regardless of the specific language used.

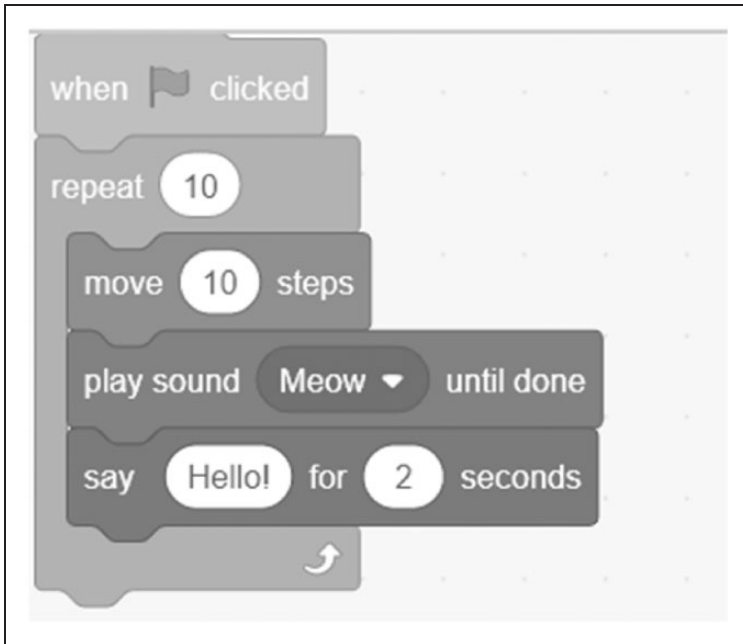
Despite the fundamental difference between these two purposes for teaching programming, a BBL could be a reasonable choice for a programming course with either purpose, with the caveat that preparing future programmers would require an eventual transition to an industry-standard language.

### *Block-Based Languages*

Block-based languages (BBLs) are a subcategory of visual programming languages. Students create programs by intuitively dragging and dropping predefined, block-shaped commands into stacks; writing lines of code is not required.

Commands of a type, such as control structures, are represented with the same-shaped block, with the shapes ensuring that command blocks can only be stacked in syntactically correct orders. BBL programs are interactive, with graphical user interfaces; for example, users often program simple video games or animations. BBLs are theorized to be easier to learn than text-based languages because they reduce the cognitive load required to learn complicated syntax, thus freeing up cognition for learning programming concepts (Bau et al., 2017). BBLs have a “low bar” for entry: Beginners can create basic animations with minimal instruction, and trial and error is a feasible route to a working program. The experience of quick success can motivate continued effort (Resnick et al., 2009). Young children can become proficient in, and excited by, BBLs, which could help recruit new generations of CS majors. And eliminating the frustration of the inevitable and ubiquitous syntax errors of text-based programming could remove a suspected cause of retention problems in computer-science courses (Besana et al., 2006; Kay et al., 2000; Kaucic & Asic, 2011).

Scratch, a web-based, open-source language, is one of the most commonly used BBLs (Lamb & Johnson, 2011; Lye & Koh, 2014). With Scratch, students intuitively drag and drop the block commands into stacks, as shown in Figure 1,

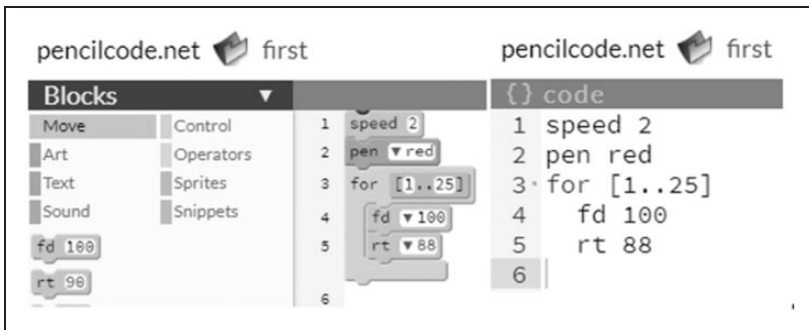


**Figure 1.** Example of Scratch Interface.

to create animations of a “sprite”—a character the student selects or designs. Programs can be quickly run without compiling, by clicking on the flags (as shown in Figure 1), allowing for immediate testing and viewing of results. Motivation and learning with Scratch are enhanced by a large online community who shares animations on the Scratch website and can explore, borrow, and modify each other’s programs (Brennan et al., 2010).

Although Scratch has the advantage of using predefined command blocks that save students from having to type code, the limited number of block types and the inability to manually edit code renders Scratch a relatively inflexible language. In response, hybrid programming languages have emerged as the next generation of visual tools for K-12 programming (Bau et al., 2017; Weintrop & Holbert, 2017). Hybrid languages offer all the advantages of earlier BBLs but add the ability to view (and edit) the block commands in the form of auto-generated, text-based code. Pencil Code (pencilcode.net) and Tynker (tynker.com) are examples of hybrid BBLs. In Pencil Code, an open-source, hybrid BBL, students toggle between block-view and code-view windows. They create programs by dragging and stacking blocks in the block-view window or editing auto-generated code in the code-view window, or both (see Figure 2).

BBLs have been used in educational settings where the goal is to develop computational thinking or other general problem-solving skills, which BBLs are presumed to engender. BBLs are also used in courses aimed at preparing programmers, to facilitate the learning of an industry-standard (target) language such as Java. Here, the presumption is that the time invested in teaching and learning the BBL is a worthy tradeoff for time that could be spent directly teaching the target language, even though learning the BBL is not the eventual goal (de Kereki, 2008; Weintrop & Holbert, 2017). Because this study focuses on this latter purpose (preparation to learn Java), we now discuss two types of rationale—affective and cognitive—that support this tradeoff.



**Figure 2.** Example of Pencil Code interface.

Note. Block view appears on the left side; code view appears on the right side.

### *Affective Rationales for Using BBLs to Prepare Programmers*

Affective rationales have been used to support the idea of starting with a BBL even when the goal is to eventually learn an industry-standard language. BBLs are designed to motivate students, leveraging the inherent fun of creating animations to excite students about programming. Motivation has long been known to promote learning, provided the motivation is intrinsic (e.g., due to enjoyment or interest) as opposed to extrinsic (e.g., for the reward of a good grade) (National Research Council, 2000; Pintrich, 1999); this has been shown to be true of programming in particular (Bergin & Reilly, 2005).

In addition to being motivational, BBLs are designed to reduce anxiety and build self-efficacy (Malan & Leitner, 2007; Meerbaum-Salant et al., 2013; Price & Barnes, 2015). Much literature demonstrates the negative impact of anxiety on learning (Derakshen & Eysenck, 2009; Wood, 2006). Computer anxiety—a fear of computers when using a computer or considering the possibility of computer use (Chua et al., 1999)—is a major and widely recognized problem for students and other adults (Brosnan, 1998; Wilfong, 2006). Computer anxiety can cause individuals to avoid computers and computer courses, but it can be mitigated through training (Chua et al., 1999). Programming anxiety is a specific case of computer anxiety that plagues many CS students and has been shown to cause poor performance and negatively impact retention in CS programs (Connolly et al., 2009; Nolan & Bergin, 2016). Theorized causes of programming anxiety include the inherent difficulty of programming, the lack of prior programming experience many students bring to introductory college CS courses, intimidation by the constant negative feedback of error messages, and the public nature of computer labs, which invites peer comparison (Connolly et al., 2009; Nolan & Bergin, 2016). Programming anxiety has only intensified over the past few decades as introductory programming courses have become more abstract and conceptual (Connolly et al., 2009).

Closely related to anxiety is self-efficacy—a judgment of “how well one can execute courses of action required to deal with prospective situations” (Bandura, 1982, p. 122), in this case in programming (Nolan & Bergin, 2016). Low programming self-efficacy correlates with reduced performance in CS courses (Bergin & Reilly, 2005), and self-efficacy has been found to mediate the relationship between anxiety and performance (Brosnan, 1998; Wilfong, 2006). As Brosnan (1998) demonstrated, only if a student’s programming anxiety reduces her programming self-efficacy will her performance be compromised; similarly, only if a reduction in programming anxiety is accompanied by an increase in programming self-efficacy will performance be improved. These findings suggest that CS educators should focus directly on building programming self-efficacy rather than just aiming to reduce anxiety. With regard to BBLs, the “low bar” for entry (the ease with which a novice can create a simple program) and the quick and continued success that BBLs offer new programmers could be

expected to develop programming self-efficacy, therefore enabling and motivating students to learn more challenging programming languages in the future.

Some research indicates that BBLs have produced positive affective outcomes with college students. Small-scale studies report that college students find Scratch to be motivating and enjoyable (Chang, 2014; de Kereki, 2008; Krafft et al., 2020; Malan & Leitner, 2007; Rizvi & Humphries, 2012) and that it can help build programming self-efficacy (Krafft et al., 2020; Rizvi & Humphries, 2012). Yet, a large-scale study of college students by Chen et al. (2019) found that prior BBL experience was no likelier to improve attitudes towards programming than was prior experience with textual languages. Few studies have examined the impact of *hybrid* BBLs in particular on programming anxiety or self-efficacy. High school students studied by Weintrop and Wilensky (2015) found that Snap!, a BBL that can be configured into a hybrid mode, was easier to read and program in than Java. Weintrop and Holbert (2017) observed high-school and graduate-school students using Pencil Code. Both groups of students chose mainly to work with the block commands, though they toggled to code view with gradually increasing frequency as they gained experience. That this hybrid BBL allows students to control their learning experience and choose their own level of support might serve to increase self-efficacy. Blanchard et al. (2019) found that 8th graders who transitioned to the use of a hybrid language after using a BBL and before transitioning to a text-only environment had more positive perceptions about the text-based environment than their counterparts who either transitioned from the BBL directly to the text-only environment or worked in the text-only environment for the entire study period. Deng et al. (2015), similarly, found that high school students who learned Pencil Code were more confident about programming than those who learned in a text-based environment.

### *Cognitive Rationale for Using BBLs to Prepare Programmers*

Learning to program requires a solid understanding of key programming concepts, such as variables and control structures. It also requires learners to construct mental models of programs (Robins et al., 2003). The cognitive rationale for having students first learn a BBL, even when the ultimate goal is to learn a text-based language, is that these cognitive accomplishments are presumed to be achieved equally well in a BBL as in the target language (Bau et al., 2017; Kaucic & Asic, 2011; Price & Barnes, 2015). But without the interference of having simultaneously to learn the complicated syntax of the target language, these critical cognitive accomplishments are presumed to occur more quickly and deeply in a BBL. Thus, when BBL-trained students transition to the target language, they learn it more rapidly than students who had been taught these concepts and skills in the target language from the start.



This cognitive rationale depends on the phenomenon of learning transfer. We next discuss two theoretical perspectives on transfer.

*The Traditional View of Transfer.* In the traditional view of transfer, a learner directly applies what she has learned in one setting (such as a school course) to another (a different course or an out-of-school situation). To use a programming example, in traditional transfer, the student who learned general programming concepts and mental models via a BBL would directly apply them to a new language, such as Java, being able to understand and code in the new language without additional assistance or training. Historically, direct transfer has been notoriously difficult to document (Bransford & Schwartz, 1999). Furthermore, in many situations, including teaching programming, it is a higher bar than educators are interested in.

*Preparation for Future Learning.* A more recent view of transfer—preparation for future learning (PFL) (Bransford & Schwartz, 1999; Schwartz et al., 2005)—resolves many of the problems with the traditional view. In the PFL view of transfer, the outcome of interest is an increased *readiness to learn* new knowledge. Learning activities that do not result in traditional (directly applicative) transfer may nevertheless build the learner's potential for further learning, a phenomenon that Schwartz and his colleagues have empirically demonstrated in multiple school activities (Schwartz et al., 2005). The PFL view of transfer better reflects the interests of college CS educators. While their desired outcome is for students to be able to program in an industry-standard, target language such as Java, they do not expect a BBL to enable students to program in the target language automatically, without any instruction in it. CS instructors of course intend to teach and provide opportunities for students to practice the target language. Their hope for teaching a BBL prior to the target language is that it will better prepare students to learn the target language—more deeply and more easily—when instruction in the target language is ultimately offered (Han Koh et al., 2010; Weintrop & Holbert, 2017).

A few studies have examined whether BBLs prepare college-level students to learn Java. Malan and Leitner (2007) found that teaching Scratch in a summer-school setting helped college students learn Java, presumably because of the key programming concepts learned via Scratch. Studies by Moskal et al. (2004) and Dann et al. (2012) document the advantage enjoyed by college students who first learned Alice over those who did not, when learning Java. Moskal et al. (2004) also found that taking a supplemental course in Alice improved the likelihood of CS1 students continuing into CS2. In contrast, de Kereki (2008) found no learning benefits in a CS1 course when Scratch was used as preparation for learning Java; the time spent learning an extra language was inferred to have mitigated any pedagogical advantage over learning Java only.



Hybrid BBLs, such as Pencil Code and Tynker, could theoretically prepare students to learn Java or other industry-standard languages even better than could a block-only language (a non-hybrid BBL) like Scratch. The ability to view block commands in auto-generated code, and to edit that code, would seem to offer a smoother transition to text-based programming (Bau et al., 2017; Weintrop & Holbert, 2017). Two studies, one by Blanchard et al. (2019) and the other by Alrubaye et al. (2019), suggest that learning a hybrid language, for 8<sup>th</sup> graders and undergraduates, respectively, can facilitate the subsequent learning of a simple text-based language. At the college level, Blanchard et al. (2020) found that the use of a hybrid BBL in a CS1 course benefited students on course assessments of reading and writing Java code but not on a validated assessment of programming concepts, compared to students who received solely text-based instruction.

### *Overview of Study*

Our study aimed to understand the benefits of two BBLs—one non-hybrid, Scratch, and one hybrid, Pencil Code—for preparing students in an introductory-level CS course for future learning of an industry-standard language, Java. We examined both the affective and cognitive influences of these languages by asking these research questions:

- RQ 1 How do students in an introductory-level CS course perceive the experience of learning an industry-standard language after learning a non-hybrid block-based programming language (BBL), a hybrid BBL, or no BBL?
- RQ 2 Does learning a non-BBL or hybrid BBL prepare students to learn an industry-standard language better than does not learning a BBL?

Three sections of a CS0 course were introduced to Java programming. One section was first taught Scratch prior to being introduced to Java; one section was first taught Pencil Code prior to Java; and the remaining section was only taught Java. RQ 1 was addressed via questionnaires to students regarding their perceptions of the ease of learning of Scratch, Pencil Code, and/or Java as a way of measuring their self-efficacy in these languages. RQ 2 was addressed with unit tests from the CS0 course that assessed the students' learning of programming concepts and Java coding.

## **Method**

### *Participants and Setting*

California State University, Northridge (CSUN), is a large, public institution in southern California that, by state policy, accepts students from the top third of

their high-school classes, prioritizing local applicants. The student body is highly diverse, and CSUN is a designated Hispanic Serving Institution, Asian American and Native American Pacific Islander Serving Institution, and Minority Institution. This study took place in 3 CSUN CS0 course sections during the Fall 2019 semester, all taught by the first author. This CS0 course is recommended, but not required, for incoming computer science and engineering majors with limited computing experience. It focuses on developing skills in problem solving, algorithm development, critical thinking, and programming, mainly in the context of computer science. The course also provides an orientation to the University and addresses campus resources, study skills, and career awareness. The department has chosen Java as the target language for its CS1 course. Because one potential benefit of CS0 is to help students transition into CS1, the first author decided to use Java as the target language for her CS0 course. Another reason for her choice of Java over a more user-friendly language was that some important course concepts could only be taught via an object-oriented language.

Data were collected from all 84 students across the 3 CS0 sections. The students completed a questionnaire that asked basic demographic information. By self-report, 78% of the participating students were freshman, and 80% were male. Computer science was the anticipated major for 62% of these students, with another 24% intending to major in a computer-related area. The questionnaire also asked the students to rate their prior level of experience with Scratch, Pencil Code, Java, and any other programming language. Most students reported at least “A little” prior experience with some programming language, and about half cited at least “Some,” but only a few had “Quite a bit” or “A lot” of experience in any language. As a group, the students had less prior experience in Pencil Code than in Scratch or Java.

This level of prior programming experience is somewhat surprising for a CS0 course, which presumes no prior experience. We can only guess why students with prior programming experience would have self-selected into this course: We suspect these students either lacked the math-course prerequisites for our university’s CS1 course, had scheduling conflicts with CS1, or underestimated their qualification for CS1. The range of prior experience in our participants offered a built-in advantage for our study, however, in that it allowed us to explore whether the BBLs were differently effective in preparing more and less experienced programmers for learning Java.

### ***Procedures***

***Languages.*** Java is the department-determined target language for the CS1 course and so was the first author’s target language choice for these CS0 sections. For this study, two BBLs were selected: one non-hybrid (Scratch) and one hybrid (Pencil Code). These two were selected among other possible BBLs

because they were free and web-based, which allowed students to run them on any device. When students toggle to code view in Pencil Code, they are offered four language choices: CoffeeScript, JavaScript, CSS, and HTML. Students in this study were instructed to select JavaScript, because it was the closest of the four languages to Java. Another popular hybrid BBL, Alice, uses Java in its code view, which might have made Alice a better choice for a course that aimed ultimately to teach Java. In Alice, however, students program 3-D animations, which we felt presented an unnecessary degree of complexity for this study. And because we wanted to minimize the differences (other than hybridity) between the two BBLs for a fairer comparison, it made sense to select the 2-D Pencil Code to compare with 2-D Scratch.

**Conditions.** The 84 students self-enrolled into one of three, separately taught sections of CS0 taught by the first author, who randomly designated one section as the Scratch section, one as the Pencil Code section, and one as the Java section; these designations were unknown to students when they enrolled. A limitation of this study is that the students were not assigned to sections randomly, but we can think of no reason why the different sections might systematically have sorted the students on any relevant factor. All sections met during the daytime, with two sections meeting twice a week, Mondays and Wednesdays, and the third meeting Fridays for twice as long. Indeed, Table 1 shows that the students in the three sections were reasonably equivalent in terms of year, gender, major, and prior programming experience, with the exception that the Java section had slightly more males and slightly more students entering the course with self-reported prior Java experience but slightly fewer students reporting at least “Some” prior experience in any language.

**Pretesting.** At the start of the semester, students in all three sections took two pretests (see Online Appendix A for sample questions) as a means of establishing a baseline level of programming knowledge:

- A Java pretest asked students to read and interpret a simple Java program and answer ten questions about what output the program would produce at various points. Scores ranged from 0 (no correct answers) to 10 (all correct answers).
- A programming concepts quiz asked students questions about basic programming concepts such as variables and loops; questions were written in English and involved no computer language. Scores ranges from 0 to 20.

Table 2 shows the results of these two pretests for each section. For each pretest, a one-way ANOVA was conducted to compare the three sections. No significant differences were found (for the Java pretest,  $F(2, 78)=0.12$ ,  $p=0.887$ ; for the concepts quiz,  $F(2, 68)=2.49$ ,  $p=0.091$ ), suggesting that the

**Table 1.** Participant Characteristics.

Characteristic	Scratch section		Pencil code section		Java section	
	<i>n</i>	%	<i>n</i>	%	<i>n</i>	%
Gender						
Male	20	71.5	22	79	22	85
Female	8	29	6	21	4	15
Year						
Year 1	20	71	22	79	20	77
Year 2	5	18	1	4	2	8
Year 3	3	11	3	11	1	4
Year 4	0	0	0	0	1	4
Post BA	0	0	0	0	1	4
No response	2	7	2	7	1	4
Anticipated major						
Computer science	17	61	15	54	17	65
Computer related	8	29	6	21	5	19
Other STEM	0	0	4	14	0	0
Non-STEM	3	11	1	4	3	12
No response	2	7	2	7	1	4
Prior Java experience						
None	16	57	16	57	13	50
A little	4	14	2	7	6	23
Some	5	18	1	4	1	4
Quite a bit	1	4	5	18	3	12
A lot	2	7	2	7	1	4
No response	2	7	1	4	1	4
Prior scratch experience						
None	16	57	18	64	13	50
A little	1	4	2	7	3	12
Some	5	18	2	7	3	12
Quite a bit	2	7	3	11	2	8
A lot	4	14	1	4	3	12
No response	2	7	1	4	1	4
Prior pencil code experience						
None	25	89	20	71	22	85
A little	2	7	2	7	0	0
Some	0	0	1	4	0	0
Quite a bit	0	0	1	4	1	4
A lot	1	4	2	7	1	4
No response	2	7	1	4	1	4
Prior experience in other language						
None	19	68	14	50	13	50
A little	5	18	3	11	6	23

(continued)

**Table 1.** Continued.

Characteristic	Scratch section		Pencil code section		Java section	
	<i>n</i>	%	<i>n</i>	%	<i>n</i>	%
Some	4	14	4	14	1	4
Quite a bit	0	0	2	7	3	12
A lot	0	0	3	11	1	4
No response	2	7	1	4	1	4
"Some" or more experience in any language	15	54	14	54	12	50

**Table 2.** Pretest Scores.

Outcome measure	Scratch section		Pencil Code section		Java section		<i>F</i>	<i>p</i>	$\eta^2$
	<i>M</i>	<i>SD</i>	<i>M</i>	<i>SD</i>	<i>M</i>	<i>SD</i>			
Java pretest	3.44	1.65	3.64	2.13	3.69	2.06	0.120	.887	0.003
Programming concept quiz	17.09	1.62	17.79	1.11	16.95	1.65	2.488	.091	0.068

Note. No *F*-test was statistically significant at  $p < .05$ .

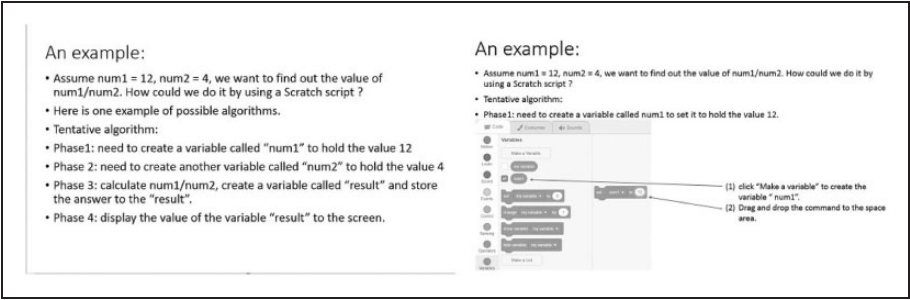
sections were equivalent in terms of prior knowledge of basic programming concepts and Java. On average, the students had a fairly strong prior knowledge of basic programming concepts but weak knowledge of Java.

**Instruction.** In the first three weeks of the semester, students in all sections received the same introduction to the concepts of personal computers, data storage, basic machine codes, and algorithms. The remainder of the course addressed four major topics: variables, selections, loops, and functions. Each of these four units included individual assignments, a group project, and a unit test. In the Scratch section, Units 1 and 2 were taught with Scratch and Units 3 and 4 were taught with Java. In the Pencil Code section, Units 1 and 2 were taught with Pencil Code and Units 3 and 4 were taught with Java. In the Java section, all four Units were taught with Java. The instructor based this configuration on two considerations: She wanted students to have enough experience with the BBL to reap its pedagogical benefits, and she knew that Units 3 and 4 involved more challenging topics and so wanted students to be able to devote their time to learning them in a single language. This instructional sequence is shown in Table 3.

To the degree possible, all assignments, projects, and tests were identical across sections except for the difference in language. An example of how a topic was addressed follows.

**Table 3.** Instructional Sequence.

Unit (Weeks)	Topic	Language of instruction		
		Scratch section	Pencil code section	Java section
Unit 1 (Weeks 4–7)	Variables	Scratch	Pencil code	Java
Unit 2 (Weeks 8–9)	Selections	Scratch	Pencil code	Java
Unit 3 (Weeks 10–12)	Loops	Java	Java	Java
Unit 4 (Weeks 13–16)	Functions	Java	Java	Java



**Figure 3.** Slides from an Early Lesson in Scratch.


In the Scratch section, to start Unit 1, the instructor explained the concept of variables and common numerical operations and showed the students how to create variables, mathematical expressions, and short programs in Scratch. Figures 3 and 4 show examples of the instructor-prepared slides that accompanied this early Scratch lesson. The instructor first displayed a possible algorithm for division, with intermediate phases written in English (left side of Figure 3). Then she showed how to translate Phase 1 from English into its corresponding Scratch command in the following slide (right side of Figure 3). She continued to translate all phases into their Scratch commands in subsequent slides. Finally, she displayed the complete translation of the algorithm in Scratch as a review (Figure 4) and prompted students to build the program by dragging and dropping the blocks.

In subsequent lessons, students practiced writing short Scratch programs from given English algorithms, then began to write their own algorithms prior to coding. Early lessons in the Pencil Code and Java sections proceeded similarly.

For Unit 3 (loops), the BBL sections transitioned to Java for the first time. The instructor leveraged the students' knowledge of the BBL by first demonstrating a program with a loop in the BBL, then translating from BBL

### An example:

- You just finished a Scratch program !
- Let's review.
- Assume number1 = 12, number2 = 4, we want to find out the value of number1/number2.
- Tentative algorithm:
- Phase1: need to create a variable called num1 to hold the value 12
- Phase 2: need to create another variable called num2 to hold the value 4
- Phase 3: calculate number1/number2, create a variable called result and store the answer to result.
- Phase 4: display the result to the screen



The Scratch code blocks are: 'when clicked' (trigger), 'set num1 to 12', 'set num2 to 4', 'set result to num1 / num2', and 'say result for 2 seconds'. Arrows from the text on the left point to these blocks: Phase 1 points to 'set num1 to 12', Phase 2 points to 'set num2 to 4', Phase 3 points to 'set result to num1 / num2', and Phase 4 points to 'say result for 2 seconds'.

Figure 4. Example Exercise in Scratch.

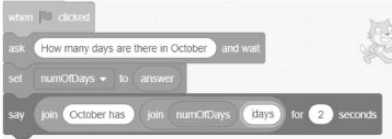
### Complete Code

```
import java.util.*;

public class PrintOutDays{
    public static void main (String[] args){
        Scanner input = new Scanner(System.in);
        System.out.println("How many days are there in October");
        int numOfDays = input.nextInt();
        System.out.println("October has " + numOfDays + "days");

        }// end of main method
    }//end of class

    Because this program has a class named " PrintOutDays", this file
    also must have the same name as its class => PrintOutDays.java
```



The Scratch code blocks are: 'when clicked' (trigger), 'ask "How many days are there in October" and wait', 'set numOfDays to answer', and 'say "October has " + numOfDays + " days" for 2 seconds'. Arrows from the text on the left point to these blocks: 'Scanner input = new Scanner(System.in);' points to 'ask', 'System.out.println("How many days are there in October");' points to 'set numOfDays to answer', and 'int numOfDays = input.nextInt();' points to 'say'.

Figure 5. Example Slide Demonstrating the Translation of Scratch Commands to Java.

commands to Java commands. Figure 5 shows an example slide that the instructor displayed to show the Scratch section how Scratch commands translated to Java. In comparison, students in the Java section only needed to be taught loop-related commands, as they had been using Java through the first two units.

### Data Collection and Analysis for RQ1

To answer RQ 1—How do students in an introductory-level CS course perceive the experience of learning an industry-standard language after learning a



non-hybrid block-based programming language (BBL), a hybrid BBL, or no BBL?—the students completed a brief, anonymous, Perceptions Questionnaire at the end of the semester, via the online course-management platform. It asked:

- How challenging was it for you to learn [Scratch/Pencil Code/Java] this semester? (10-point Likert scale; students asked only about the language[s] taught in their section)
- How confident do you feel now about your ability to program in [Scratch/Pencil Code/Java]? (10-point Likert scale; students asked only about the language[s] taught in their section)
- Do you think learning [Scratch/Pencil Code] first made it easier for you to learn Java? (10-point Likert scale; students asked only about the language taught in their section; not asked in Java section)

Responses for each question were disaggregated by group and descriptive statistics were calculated. Items asking about the BBLs were only answered by two groups: the Scratch and Pencil Code groups; for these items, independent *t*-tests were conducted to determine the significance of the differences between the groups' ratings. The other items were answered by all three groups; for these items, one-way ANOVAs or ANCOVAs were conducted.

This questionnaire also asked:

If you feel learning [Scratch/Pencil Code] first made it easier for you to learn Java, please explain why. (Write-in response, not asked in Java section).

Students wrote short explanations, which we grouped into categories; we then tallied the frequency of responses in each category.

### *Data Collection and Analysis for RQ2*

To answer RQ 2—Does learning a non-hybrid BBL or hybrid BBL prepare students to learn an industry-standard language better than does not learning a BBL?—we examined the scores on the final two unit tests. All students, regardless of section, took these same two unit tests, which assessed their knowledge of Java. Multiple-choice and short-answer questions asked students to predict the output of sections of Java code; other questions required students to write Java code to complete incomplete programs or to create entire programs (sample tasks from these unit tests can be found in Online Appendix B). The instructor evaluated these tests for completeness and correctness of coding. In total, these tests constituted 40% of the students' course grade.

The means and standard deviations of the scores on each unit test were calculated for each of the three groups and compared. A one-way ANOVA was conducted to determine the significance of any differences among the groups' means.

Because some students had brought prior Java experience to the course, we were interested to see whether RQ2 might be answered differently for students with different degrees of prior Java knowledge. For example, we might predict that learning a BBL before learning Java might be more helpful for students with little initial knowledge of Java but less useful to students with more initial Java knowledge. To explore this idea, we ran an ANCOVA to test for differences among the unit test results for the three groups while controlling for scores on the Java pretest. A univariate ANOVA was first used to confirm the required assumptions of ANCOVA.

Results

RQ1: Student Perceptions

*Difficulty and Confidence With Each Language.* Table 4 shows the mean ratings that students in each section gave for how challenging it was to learn each language and their confidence, by the end of the course, in programming in that language.

In general, students who learned either of the BBLs found the BBL easier to learn than Java, as would be expected, given that these BBLs were designed to be easier to learn ( $p$ -values  $< 0.001$ ). Students in the Scratch group were also more confident about their ability to program in Scratch than in Java. ( $p < 0.001$ ), but the Pencil Code group did not feel differently confident about programming in Pencil Code and Java. Independent  $t$ -tests showed no significant difference between Scratch and Pencil Code in terms of how easy students

Table 4. Mean Difficulty and Confidence Ratings.

Outcome measure	Scratch		Pencil Code		Java		$F/t$	$p$	$\eta^2 /$ Cohen's $d$
	$M$	$SD$	$M$	$SD$	$M$	$SD$			
Perceived difficulty of learning Java	−7.57	2.86	6.37	3.00	5.72	2.30	3.12	.050	0.075
Perceived confidence in Java	5.29	2.32	5.52	3.01	6.96	1.79	3.58*	.033	0.085
Perceived difficulty of learning BBL	2.96	2.01	3.88	3.36			−1.23	.224	.33
Perceived confidence in BBL	8.25	2.03	7.23	2.97			1.48	.145	.40
Helpfulness of BBL for learning Java	6.56	2.72	4.62	3.29			2.34	.023	.64

Note. The first two outcomes were tested with a one-way ANOVA, and the second three with an independent  $t$ -test.

\* $p < .05$ .

felt these languages were to learn ( $t = -1.231, p = 0.224$ ) or how confident students were in using them ( $t = 1.481, p = 0.154$ ).

More important are the perceptions across groups about learning Java. We conducted one-way ANOVAs to test differences among the three groups in terms of their perceived difficulty in learning Java and their confidence in Java. There was a significant omnibus effect for Java confidence ( $F[2, 77] = 3.58, p = 0.033$ ) but not for difficulty ( $F[2, 77] = 3.12, p = 0.050$ ). Tukey HSD tests showed a difference in only one mean comparison: students in the Scratch section found learning Java more difficult ( $M = 7.57$ ) than did students in the Java section ( $M = 5.72, p = 0.044$ ). The interpretation is that the students who learned only Java for the entire semester felt that learning Java was easier than did the students who spent the first part of the semester learning Scratch (but not Pencil Code) before being introduced to Java. Similarly, students in the Java section felt more confident ( $M = 6.96$ ) about programming in Java than did the Scratch section ( $M = 5.29, p = 0.039$ ), but not significantly more than did the Pencil Code section. Overall, these data suggest that, while Scratch and Pencil Code are easier to learn than Java, spending time learning them did not pay off in terms of students' feelings of ease or confidence in learning the target language, Java.

When directly asked whether learning the BBLs made learning Java easier, students in the Scratch section gave Scratch a mildly positive rating ( $M = 6.56$  on a scale of 1 [Not easier] to 10 [Much easier]) but students in the Pencil Code section gave a mildly negative rating ( $M = 4.62$ ), a difference that was significant ( $t = 2.344, p = 0.023$ ). This is a somewhat surprising finding: Pencil Code, as a hybrid between a BBL and code-based language, would seem to provide a smoother transition to Java than would Scratch, which offers no code-writing practice.

*How the BBL Helped Students Learn Java.* Students who felt that learning a BBL had helped them learn Java were asked to briefly explain why they felt this way. Table 5 shows the categories of reasons and the frequencies of responses within each.

**Table 5.** Reasons Why the BBL Helped Students Learn Java.

Reason categories	Scratch section	Pencil Code section
BBL provided a basic understanding of programming concepts	7	3
The blocks simplified programming	4	0
BBL had a close relation to Java	0	3
BBL and Java had a similar structure	1	2
BBL eliminated memorization or complicated syntax	2	0
BBL was more visual	1	0
Neutral response (e.g., "BBL was helpful")	1	4

According to these students, the main value of the BBLs was that they helped students understand basic programming concepts, which may have prepared them indirectly to learn Java, by giving them a conceptual understanding of what a Java program was “doing.” Less cited were more direct links between the BBL and Java, though, as expected, such reasons were more prevalent in the Pencil Code section. Interestingly, only students in the Scratch section cited the simplicity or visual nature of the blocks or the elimination of complicated syntax, even though Pencil Code offers these same features.

An important question is whether students in the Pencil Code section actually took advantage of the hybrid nature of Pencil Code. If students never toggled to Pencil Code’s code-view window and edited or wrote code, they were for all practical purposes using a non-hybrid BBL—that is, only manipulating blocks. In the end-of-course Perceptions Survey, students in the Pencil Code section were asked whether they used both views (block and code) or exclusively worked in one view. Of the 26 Pencil Code students who responded to this question, 20 indicated that they used both views, presumably practicing some code-writing or editing, while only four stayed exclusively in block view and two in code view. Because most Pencil Code students used both block and code view, it is surprising that so few of them cited this transitional experience to coding as a reason why Pencil Code helped them learn Java.

RQ 2: Preparation for Learning Java

*Overall Java Performance.* Table 6 shows the mean scores on the two unit tests on Java.

ANOVAs showed no differences among the three groups on either Java unit test (for the first Java unit test:  $F[2, 74] = 0.018, p = 0.982$ ; for the second Java unit test:  $F[2, 80] = 1.373, p = 0.259$ ). On average, then, spending time on Scratch or Pencil Code appears not to have harmed students’ learning of Java, despite those students spending less time learning Java than students in the Java section, but it also does not appear to have prepared them any better for learning Java.

**Table 6.** Scores on Java Unit Tests Without Controlling for Pretest Scores.

Outcome measure	Scratch		Pencil Code		Java		<i>F</i>	<i>p</i>	$\eta^2$
	<i>M</i>	<i>SD</i>	<i>M</i>	<i>SD</i>	<i>M</i>	<i>SD</i>			
Scores on Java unit test 1	4.19	2.69	4.328	2.6161	4.315	2.7705	0.018	.982	0.000
Scores on Java unit test 2	7.185	4.17	8.517	3.21	8.593	3.1288	1.373	.259	0.033

Note. No *F*-test was statistically significant at  $p < .05$ .

*Java Performance Controlled for Prior Java Knowledge.* Prior to running the ANCOVA to determine whether prior Java knowledge mediated the effect of the BBLs on Java performance, we used a univariate ANOVA to confirm the required assumptions for ANCOVA. Tables 7 and 8 show the results. This ANOVA indicated that the required assumptions were met: There was no difference in Java pretest scores among the three groups at the  $p < 0.05$  level ( $p = 0.887$ ) and there was homogeneity of regression ( $p = 0.063$ ). As shown in Table 7, the ANCOVA showed that, even when controlling for Java pretest scores, there were no differences among the three groups on either unit test (for the first Java unit test,  $F[2, 71] = 0.02$ ,  $p = 0.981$ ; for the second Java unit test,  $F[2, 75] = 1.04$ ,  $p = 0.357$ ). Thus, while prior Java knowledge was helpful to students

**Table 7.** Scores on Java Unit Test 1 After Controlling for Java Pretest Score.

Source	Type III Sum of Squares	df	Mean Square	F	p	Partial $\eta^2$
Corrected Model	82.11	3	27.37	4.56**	.006	.162
Intercept	74.19	1	74.19	12.37**	.001	.148
Pretest score	82.10	1	82.10	13.69***	< .001	.162
Groups	0.23	2	0.11	0.02	.981	.001
Error	425.78	71	6.00			
Total	1822.50	75				
Corrected Total	507.89	74				

$R^2 = .16$  (Adjusted  $R^2 = .13$ ).

\*\* $p < 0.01$ .

\*\*\* $p < 0.001$ .

**Table 8.** Scores on Java Unit Test 2 After Controlling for Java Pretest Score.

Source	Type III Sum of Squares	df	Mean Square	F	p	Partial $\eta^2$
Corrected Model	105.17	3	35.06	3.12*	.031	.111
Intercept	648.55	1	648.55	57.79***	< .001	.435
Pretest score	79.42	1	79.42	7.08*	.010	.086
Groups	23.41	2	11.71	1.04	.357	.027
Error	841.72	75	11.22			
Total	6051.00	79				
Corrected Total	946.89	78				

$R^2 = .11$  (Adjusted  $R^2 = .08$ ).

\* $p < 0.05$ .

\*\*\* $p < 0.001$ .

on their Java unit tests, it appears that the level of prior Java knowledge did not mediate the impact of learning a BBL.

## Discussion and Conclusions

In this study, we investigated the degree to which learning a BBL (Scratch or Pencil Code) in a CS0 course prepared college students for learning Java later in the course, compared to learning only Java throughout the course. Both student perceptions and actual Java performance were examined. Overall, the results show no advantage to learning a BBL, and weak evidence of possible disadvantage.

While students who learned a BBL found it easier to learn than Java, they found learning Java more difficult than did the students who had learned Java from the start of the course, and by the end of the course, the BBL students felt less confident in Java programming than the Java-only group (although these differences were only statistically significant for the Scratch group). Many students (especially in the Scratch group) perceived that learning the BBL had helped them learn Java, but their perceived competency in Java at the end of the course indicated otherwise. A hope for CS0 courses is that students will leave feeling motivated and confident about taking future CS courses and majoring in CS. These findings suggest that taking time away from Java learning at the start of a CS0 course in order to learn a BBL may work against this hope, rendering students less confident about programming in a language that is central to future CS courses.

Paralleling these perceptual findings, the students' unit test scores showed no impact, positive or negative, of first learning a BBL on learning Java. Controlling for the students' level of prior Java knowledge did not change this finding; learning a BBL was equally ineffective as preparation for learning Java regardless of the student's prior Java knowledge.

The hybridity of Pencil Code did not offer the hypothesized advantage. Even though individual students in the Pencil Code section used both its block and editing views, their outcomes were no better than those of the students who only worked with blocks (Scratch) or text (Java).

These findings echo the results of a related study by Weintrop and Wilensky (2019), who compared two high school classes in which students learned to program in either a BBL or simple text-based (non-Java) environment for five weeks, before transitioning to Java for the remaining ten weeks of the course. Weintrop and Wilensky found no differences between the groups in Java performance, programming confidence, enjoyment, or intent to take further CS courses, leading them to conclude that, when the goal is to prepare students to learn an industry-standard text-based language, a BBL offers no advantage over a text-based language. Our study makes additional contributions by examining a different population—college students, by adding a hybrid BBL to the

comparison, and by using a “control” situation that is more realistic for a college-level course: teaching the target language from Day 1.

Like Weintrop and Wilensky, we believe that BBLs and hybrid BBLs may have important advantages that our studies do not diminish. BBLs may enhance students’ enjoyment of and confidence in programming, especially for young students and true beginners. They may also help beginning programmers grasp key programming concepts. But for college students, our study suggests that BBLs and hybrid BBLs are unnecessary as preparation for learning Java; even students in the lowest-level CS course appear ready to learn Java from the start. We note that many students in our study were not beginners but brought some prior programming experience to the course. This, however, is probably typical of first-year college students today, reflecting increasing opportunities to learn programming in K-12 classes or after-school activities (Google & Gallup, 2016). Our findings resonate somewhat with Chen et al.’s (2019) study and Hu et al.’s (2020) meta-analysis, both of which found decreased academic benefits from experience with BBLs for college students.

This study has several limitations. It examined a single instructor at one university, which restricts generalization to other settings. The small numbers of students in each group and subgroup limited statistical power. Many other instructional conditions are possible beyond the ones used here, in which a period of instruction solely in the BBL was followed by a period of instruction solely in Java. Different outcomes might result from a shift in the relative length of these periods, or from a different kind of sequence, where, for example, the BBL is revisited briefly for new topics. Because the instructor wanted to keep the instruction across sections as similar as possible, she did not take full advantage of the capacity of Scratch and Pencil Code to create videos and games, which might have compromised these BBLs’ power to develop programming confidence. The instruments that were used to measure the students’ perceptions and performances were not empirically validated; the performance measures in particular were the normal course assessments. Finally, we did not follow the students in our study into the subsequent semester to try to detect differences among the sections in terms of persistence into future CS courses or the CS major, although these are the outcomes of perhaps the greatest interest to educators.

Future research could investigate whether the use of a hybrid BBL, such as Alice, that uses Java as the default code, serves as a better transition to Java than does Pencil Code, whose default code is not Java. Another research direction stems from the observation that the students in this study, on average, entered the CS0 course with a fairly strong grasp of basic programming concepts but little Java knowledge. This suggests that CS0 students may not need the support of a BBL to understand programming concepts and may be ready to start with a text-based language. It would be fruitful to explore the benefits of



starting with a less complex and intimidating text-based language (e.g., Python or Swift), rather than a BBL, before transitioning to an industry-standard one (e.g., Java).

### Declaration of Conflicting Interests

The authors declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

### Funding

The authors received no financial support for the research, authorship, and/or publication of this article.

### ORCID iD

Wen-Chin Hsu  <https://orcid.org/0000-0001-7654-853X>

### Supplemental Material

Supplemental material for this article is available online.

### References

- Alrubaye, H., Ludi, S., & Mkaouer, M. W. (2019). *Comparison of block-based and hybrid-based programming environments in transferring programming skills to text-based environment* [Paper presentation]. 29th Annual International Conference on Computer Science and Software Engineering (pp. 100–109). IBM Corporation. <https://www.researchgate.net/publication/333701279>
- Bandura, A. (1982). Self-efficacy mechanism in human agency. *American Psychologist*, 37(2), 122–147. <https://doi.org/10.1037/0003-066X.37.2.122>
- Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communications of the ACM*, 60(6), 72–80. <https://doi.org/10.1145/3015455>
- Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2), 32–36. <https://www.researchgate.net/publication/220613179>
- Bergin, S., & Reilly, R. (2005). The influence of motivation and comfort-level on learning to program. In P. Romero, J. Good, E. Acosta Chaparro, & S. Bryant (Eds). *Proceedings of the 17th workshop of the psychology of programming interest group* (pp. 293–304). Sussex University. <http://citeserx.ist.psu.edu/viewdoc/download?doi=10.1.1.443.427&rep=rep1&type=pdf>
- Besana, G. M., Dettori, L., & Steinbach, T. A. (2006). An invitation to IT: Redesigning the first year. *Journal of Computing Sciences in Colleges*, 21(6), 130–139. <https://doi.org/10.5555/1127442.1127461>
- Blanchard, J., Gardner-McCune, C., & Anthony, L. (2019). *Effects of code representation on student perceptions and attitudes toward programming* [Paper presentation]. IEEE Symposium on Visual Languages & Human-Centric Computing (VL/HCC'2019) (pp. 127–131), Memphis, TN, USA, October 14–18, 2019. <https://init.cise>

- ufl.edu/effects-of-code-representation-on-student-perceptions-and-attitudes-toward-programming/
- Blanchard, J., Gardner-McCune, C., & Anthony, L. (2020). *Dual modality instruction and learning: A case study in CS1* [Paper presentation]. 51st ACM Technical Symposium on Computer Science Education, February 2020 (pp. 818–824). ACM. <https://doi.org/10.1145/3328778.3366865>
- Bransford, J. D., & Schwartz, D. L. (1999). Rethinking transfer: A simple proposal with multiple implications. *Review of Research in Education*, 24(1), 61–100. [https://aaalab.stanford.edu/assets/papers/earlier/Rethinking\\_transfer\\_a\\_simple\\_proposal\\_with\\_multiple\\_implications.pdf](https://aaalab.stanford.edu/assets/papers/earlier/Rethinking_transfer_a_simple_proposal_with_multiple_implications.pdf)
- Brennan, K., Monroy-Hernández, A., & Resnick, M. (2010). Making projects, making friends: Online community as catalyst for interactive media creation. *New Directions for Youth Development*, 2010(128), 75–83. <https://doi.org/10.1002/yd.377>
- Brosnan, M. J. (1998). The impact of computer anxiety and self-efficacy upon performance. *Journal of Computer Assisted Learning*, 14(3), 223–234. <https://doi.org/10.1046/j.1365-2729.1998.143059.x>
- Chang, C.-K. (2014). Effects of using Alice and scratch in an introductory programming course for corrective instruction. *Journal of Educational Computing Research*, 51(2), 185–204. <https://doi.org/10.2190/EC.51.2.c>
- Chen, C., Haduong, P., Brennan, K., Sonnert, G., & Sadler, P. (2019). The effects of first programming language on college students' computing attitude and achievement: A comparison of graphical and textual languages. *Computer Science Education*, 29(1), 23–28. <https://doi.org/10.1080/08993408.2018.1547564>
- Chua, S. L., Chen, D., & Wong, A. F. L. (1999). Computer anxiety and its correlates: A meta-analysis. *Computers in Human Behavior*, 15(5), 609–632. [https://doi.org/10.1016/S0747-5632\(99\)00039-4](https://doi.org/10.1016/S0747-5632(99)00039-4)
- Connolly, C., Murphy, E., & Moore, S. (2009). Programming anxiety amongst computing students—A key in the retention debate? *IEEE Transactions on Education*, 52(1), 52–56. <https://doi.org/10.1109/TE.2008.917193>
- Cook, C. R. (1997). *CS0 Computer science orientation course* [Symposium]. 28th SIGCSE Technical Symposium on Computer Science Education (pp. 87–91) San Jose, CA. <https://doi.org/10.1145/268084.268119>
- Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). *Mediated transfer. Alice 3 to Java* [Symposium]. 43rd ACM Technical Symposium on Computer Science Education (pp. 141–146), Raleigh, NC. <https://doi.org/10.1145/2157136.2157180>
- de Kereki, I. F. (2008). *Scratch: Applications in computer science* [Paper presentation]. 38th ASEE/IEEE Frontiers in Education Conference, Saratoga Springs, NY. IEEE (pp. T3B-7-T3B-11). <https://library.villanova.edu/Find/Record/1452836>
- Deng, W., Pi, Z., Lei, W., Zhou, Q., & Zhang, W. (2020). Pencil code improves learners' computational thinking and computer learning attitude. *Computer Applications in Engineering Education*, 28(1), 90–104. <https://doi.org/10.1002/cae.22177>
- Derakshen, N., & Eysenck, M. W. (2009). Anxiety, processing efficiency, and cognitive performance: New developments from attentional control theory. *European Psychologist*, 14(2), 168–176. <https://doi.org/10.1027/1016-9040.14.2.168>
- Google Inc. & Gallup Inc. (2016). *Trends in the State of Computer Science in U.S. K-12 Schools*. <http://goo.gl/j291E0>

- Grover, S., & Pea, R. (2013). Computational thinking in K-12: A review of the state of the field. *Educational Researcher*, 42(1), 38–43. <https://doi.org/10.3102/0013189X12463051>
- Han Koh, K., Basawapatna, A., Bennett, V., & Repenning, A. (2010). *Towards the automatic recognition of computational thinking for adaptive visual language learning* [Symposium]. IEEE International Symposium on Visual Languages and Human Centric Computing, Madrid, (pp. 59–66). IEEE. <https://doi.org/10.1109/VLHCC.2010.17>
- Hu, Y., Chen, C.-H., & Su, C.-H. (2020). Exploring the effectiveness and moderators of block-based visual programming on student learning: A meta-analysis. *Journal of Educational Computing Research*, 58(8), 1467–1493. <https://doi.org/10.1177/0735633120945935>
- Kaucic, B., & Asic, T. (2011). *Improving introductory programming with scratch?* [Paper presentation] 34th International Convention on Information and Communication Technology, Electronics, and Microelectronics (MIPRO), Opatija, Croatia (pp. 1095–1100). IEEE. <http://ieeexplore.ieee.org/document/5967218>
- Kay, J., Barg, M., Fekete, A., Greening, T., Hollands, O., Kingston, J. H., & Crawford, K. (2000). Problem-based learning for foundation computer science courses. *Computer Science Education*, 10(2), 109–128. [https://doi.org/10.1076/0899-3408\(200008\)10:2;1-C;FT109](https://doi.org/10.1076/0899-3408(200008)10:2;1-C;FT109)
- Klahr, D., & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, Learning, and transfer. *Cognitive Psychology*, 20(3), 362–404. [https://doi.org/10.1016/0010-0285\(88\)90004-7](https://doi.org/10.1016/0010-0285(88)90004-7)
- Kolling, M. (1999). The problem of teaching object-oriented programming, part 1: Languages. *Journal of Object-Oriented Programming*, 11(8), 8–15. <https://www.researchgate.net/publication/220278819>
- Krafft, M., Fraser, G., & Walkinshaw, N. (2020, June). *Motivating adult learners by introducing programming concepts with scratch* [Paper presentation]. 4th European Conference on Software Engineering Education (pp. 22–26). <https://doi.org/10.1145/3396802.3396818>
- Lamb, A., & Johnson, L. (2011). Scratch: Computer programming for 21st century learners. *Teacher Librarian*, 38(4), 64–68. <https://scholarworks.iupui.edu/bitstream/handle/1805/8622/384.pdf?sequence=1>
- Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, 41, 51–61. <https://doi.org/10.1016/j.chb.2014.09.012>
- Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. *ACM SIGCSE Bulletin*, 39(1), 223–227. <https://doi.org/10.1145/1227504.1227388>
- Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). Learning computer science concepts with scratch. *Computer Science Education*, 23(3), 239–264. <https://doi.org/10.1080/08993408.2013.832022>
- Moors, L., Luxton-Reilly, A., & Denny, P. (2018). *Transitioning from block-based to text-based programming languages* [Paper presentation]. 2018 International Conference on Learning and Teaching in Computing and Engineering (pp. 57–64). IEEE. <https://doi.org/10.1109/LaTICE.2018.000-5>

- Moskal, B., Lurie, D., & Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. *ACM SIGCSE Bulletin*, 36(1), 75–79. <https://doi.org/10.1145/1028174.971328>
- National Research Council. (2000). *How people learn: Brain, mind, experience, and school: Expanded edition*. The National Academies Press. <https://doi.org/10.17226/9853>
- Nolan, K., & Bergin, S. (2016). *The role of anxiety when learning to program. A systematic review of the literature* [Paper presentation]. 16th Koli Calling International Conference on Computing Education Research (pp. 61–70), New York, NY. <https://doi.org/10.1145/2999541.2999557>
- Pane, J. F., Ratanamahatana, C. A., & Myers, B. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2), 237–264. <https://doi.org/10.1006/ijhc.2000.0410>
- Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, 39(4), 204–223. <https://doi.org/10.1145/1345375.1345441>
- Pintrich, P. R. (1999). The role of motivation in promoting and sustaining self-regulated learning. *International Journal of Educational Research*, 31(6), 459–470. [https://doi.org/10.1016/S0883-0355\(99\)00015-4](https://doi.org/10.1016/S0883-0355(99)00015-4)
- Price, T. W., & Barnes, T. (2015). *Comparing textual and block interfaces in a novice programming environment* [Paper presentation]. *Eleventh Annual International Conference on International Computing Education Research* (pp. 91–99), Omaha, NE. <https://doi.org/10.1145/2787622.2787712>
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60–67. <https://doi.org/10.1145/1592761.1592779>
- Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137–172. <https://doi.org/10.1076/csed.13.2.137.14200>
- Rizvi, M., & Humphries, T. (2012). *A scratch-based CS0 course for at-risk computer science majors* [Paper presentation]. 2012 Frontiers in Education Conference (pp. 1–5), Seattle, WA. <https://doi.org/10.1109/FIE.2012.6462491>
- Schwartz, D. L., Bransford, J. D., & Sears, D. (2005). Efficiency and innovation in transfer. In J. Mestre (Ed.), *Transfer of learning from a modern multidisciplinary perspective* (pp. 1–51). Information Age Publishing Inc.
- Vihavainen, A., Airaksinen, J., & Watson, C. (2014). *A systematic review of approaches for teaching introductory programming and their influence on success* [Paper presentation]. 10th Annual International Conference on International Computing Education Research (pp. 19–26), Glasgow, Scotland, UK. ACM. <https://doi.org/10.1145/2632320.2632349>
- Watson, C., & Li, F. W. B. (2014). *Failure rates in introductory programming revisited* [Paper presentation]. 2014 Conference on Innovation Technology in Computer Science Education (pp. 39–44), New York, NY. ACM. <https://doi.org/10.1145/2591708.2591749>

- Weintrop, D., & Holbert, N. (2017). *From blocks to text and back. Programming patterns in a dual-modality environment* [Symposium]. 2017 ACM SIGCSE Technical Symposium on Computer Science Education (pp. 633–638), Seattle, WA. ACM. <https://doi.org/10.1145/3017680.3017707>
- Weintrop, D., & Wilensky, U. (2015). *To block or not to block, that is the question. Students' perceptions of blocks-based programming* [Paper presentation]. 14th International Conference on Interaction Design and Children (pp. 199–208), Boston, MA. <https://doi.org/10.1145/2771839.2771860>
- Weintrop, D., & Wilensky, U. (2019). Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education*, 142, 103646. <https://doi.org/10.1016/j.compedu.2019.103646>
- Wilfong, J. D. (2006). Computer anxiety and anger: The impact of computer use, computer experience, and self-efficacy beliefs. *Computers in Human Behavior*, 22(6), 1001–1011. <https://doi.org/10.1016/j.chb.2004.03.020>
- Wood, J. (2006). Effect of anxiety reduction on children's school performance and social adjustment. *Developmental Psychology*, 42(2), 345–349. <https://doi.org/10.1037/0012-1649.42.2.345>

### Author Biographies

**Wen-Chin Hsu** is an assistant professor of Computer Science at California State University, Northridge. Her research interests include computer science education, human computer interaction, and data science analyses. Within the area of computer science education, she focuses on the use of educational tools, pedagogical strategies, and technologies in introductory programming courses.

**Julie Gainsburg** is professor and chair of the Department of Secondary Education in the Michael D. Eisner College of Education at California State University, Northridge. She has studied the mathematical behavior of engineers and engineering students, mathematics-teacher education, and the use of real-world problems in mathematics education.