2026 ASEE ANNUAL CONFERENCE June 21–24, 2026 | Charlotte Convention Center, NC
Host Sponsor UNC Charlotte
ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION
Paper ID #52685

Bridging Mathematics and Programming in Introductory Computer Science: Lessons from Assignment Design and Classroom Practice

Dr. Xiaojin Ye, Farmingdale State College, SUNY

Xiaojin Ye is an Assistant Professor in the Department of Computer Systems at Farmingdale State College, SUNY, where she has served since Fall 2022. Prior to joining Farmingdale, she taught mathematics at Fordham University and Hudson County Community College. She earned her Ph.D. in Computer Science from the Graduate Center of the City University of New York. Her research interests include theoretical computer science, set operators, mathematical morphology, machine learning, mathematics education, and computer science education.

Bridging Mathematics and Programming in Introductory Computer Science: Lessons from Assignment Design and Classroom Practice

Abstract

Mathematical reasoning is fundamental to computer science (CS); however, many students entering introductory programming courses struggle to connect foundational mathematical concepts with computational logic. Drawing from over a decade of college-level mathematics teaching experience, doctoral studies in mathematical theory within computer science, and several years of teaching introductory computer science at a four-year college, this paper explores how lesson and assignment design can bridge the gap between mathematical thinking and programming practice in an introductory Java course.

This reflective paper presents classroom strategies, lesson structures, and programming assignments intentionally designed to strengthen mathematical reasoning through guided problem solving, logical analysis, and coding exercises. Three instructional strategies are emphasized: (1) making the mathematical foundations of programming constructs explicit, (2) translating real-world problem scenarios into mathematical representations prior to coding, and (3) connecting mathematical models directly to programming constructs and program state. Common student challenges, including confusion surrounding integer division in Java, misunderstandings of quotient–remainder reasoning and logical conditions, difficulties translating mathematical relationships into Boolean expressions, and errors in applying operator precedence, are discussed alongside concrete classroom examples, including a representative lab assignment that guides students from mathematical modeling to program implementation. Reflective classroom observations and informal assessments suggest that embedding mathematical reasoning throughout lessons and assignments can support improved conceptual understanding, confidence, and engagement among novice programmers. The paper concludes with lessons learned, practical recommendations for instructors, and broader implications for integrating mathematical thinking into computer science education.

Keywords

Computer Science Education; Introductory Programming; Mathematical Reasoning; Instructional Design; Java Programming; Assignment Design

Introduction

Mathematical reasoning is fundamental to computer science (CS); however, many students entering introductory programming courses struggle to connect foundational mathematical concepts with computational logic. Core programming topics such as variables, arithmetic expressions, conditional statements, loops, and algorithmic thinking rely heavily on abstraction, logical reasoning, and mathematical structure. Foundational work in computer science has long

emphasized the central role of mathematical reasoning in algorithmic design and analysis [1], and national curricular guidelines further reinforce this connection by identifying problem solving, analytical thinking, and algorithmic reasoning as essential learning outcomes of undergraduate CS programs [2]. Despite this close relationship, many novice programmers lack fluency and confidence when applying mathematical concepts in programming contexts.

Prior research in computer science education indicates that the challenges faced by novice programmers extend beyond learning syntax. Learning to program requires students to engage in abstract reasoning and computational logic that often differ from the procedural mathematics they have previously encountered [3]. Studies have shown that students may be able to read or trace code line by line yet still struggle to reason about overall program behavior and underlying logic [4]. Computational thinking has been characterized as a problem-solving process grounded in abstraction, decomposition, and algorithmic formulation [5], all of which are closely related to mathematical reasoning. Systematic reviews of introductory programming education further confirm that novice learners commonly experience difficulties related to problem solving, conceptual understanding, and applying mathematical reasoning in programming contexts [6], [7].

In introductory programming classrooms, these difficulties frequently manifest as confusion surrounding integer division in Java, misunderstandings of logical conditions, errors in applying operator precedence, and difficulty constructing correct conditional expressions. Such challenges highlight a persistent gap between students' mathematical preparation and their ability to translate mathematical reasoning into executable programs.

Prior work has highlighted the importance of reasoning, abstraction, and modeling in introductory programming and computational thinking [3], [5]; however, much of this literature remains primarily diagnostic or conceptual. In contrast, the present work contributes a practice-based instructional framework that operationalizes mathematical reasoning within everyday programming constructs in CS1 (introductory programming), illustrating how formal mathematical ideas can be made explicit, progressively developed, and directly mapped to program state and control flow.

To address this gap, this paper adopts an instructional approach that explicitly integrates mathematical reasoning into introductory programming instruction. Rather than treating mathematics as background knowledge that students are expected to apply independently, the instructional design described here treats mathematical reasoning as an explicit learning objective. This perspective aligns with prior work emphasizing the importance of connecting formal mathematical reasoning with programming instruction in introductory computer science contexts [8], as well as practice-based research focused on supporting students' reasoning development in programming courses [9]. The approach is further informed by educational research highlighting the value of making expert reasoning visible to novice learners [10] and supporting structured progression across representations [11]. It also aligns with research calling for reduced overreliance on syntax-focused instruction [3], [12] and closer alignment among learning outcomes, instructional activities, and assessment [13]. Together, these perspectives motivate a set of core design principles intended to strengthen the connection between mathematical reasoning and programming practice.

This reflective paper presents classroom strategies, lesson structures, and programming assignments designed to strengthen mathematical reasoning through guided problem solving, logical analysis, and structured coding activities in an introductory Java course at a four-year college. Instructional examples focus on commonly challenging topics, including integer division, the modulo operator, conditional logic, and operator precedence, to illustrate how mathematical abstraction is realized in programming practice. The paper concludes with lessons learned, practical recommendations for instructors, and broader implications for integrating mathematical thinking into introductory computer science education.

Instructional Context

The instructional setting for this study is Computer Programming I (CSC 111), an introductory computer science course taught in Java at Farmingdale State College (SUNY), a public four-year institution. The course serves as the first required programming course for students majoring in Computer Science and Computer Programming and Information Systems and is typically taken by first- or second-year undergraduates. Across these two majors, approximately 800 students are enrolled. The course is offered in about ten sections annually, with an average yearly enrollment of approximately 250 students.

The course introduces fundamental programming and problem-solving concepts using an object-oriented language. Core topics include variables, arithmetic expressions, conditional statements (selection), loops (repetition), methods, classes, arrays, and basic algorithmic thinking. Java is used as the instructional language due to its strong typing, widespread industry adoption, and alignment with subsequent courses in the curriculum.

The student population enrolled in the course is academically diverse. While most students have completed prerequisite mathematics coursework such as college algebra or precalculus, their levels of mathematical fluency and confidence vary widely. Some students demonstrate procedural competence in mathematics but struggle with abstraction and symbolic reasoning, while others are comfortable with mathematical concepts yet encounter difficulty applying them in programming contexts. Few students have prior experience explicitly connecting mathematical reasoning to computational problem solving. Consequently, the course represents a heterogeneous learning environment in which assumptions about students' ability to transfer mathematical knowledge to programming tasks cannot be made uniformly.

The goals of the course include developing students' ability to write correct and readable programs, reason about program behavior, and apply structured problem-solving strategies to computational tasks. Beyond learning syntax and language features, students are expected to analyze problems, design logical solutions, and understand how mathematical relationships influence program behavior. These goals provide a natural and necessary context for explicitly integrating mathematical reasoning into programming instruction.

The instructional approach described in this paper is informed by the author's academic training and teaching experience in both mathematics and computer science. The author was originally

trained in mathematics, earning a bachelor's degree and two master's degrees in pure mathematics, before transitioning into doctoral training in computer science with a focus on mathematical theory. The author has taught college-level mathematics for approximately twelve years and has taught introductory computer science programming courses full-time for the past three years.

This interdisciplinary background provides a dual perspective on student learning challenges. Experience teaching mathematics has revealed recurring difficulties students face in abstraction, symbolic manipulation, and logical reasoning. These skills are foundational to programming. Similarly, experience teaching introductory computer science has shown that many programming errors stem not solely from syntactic misunderstandings, but from incomplete or inaccurate reasoning about underlying mathematical and logical concepts.

This dual perspective motivates the explicit integration of mathematical reasoning into programming instruction described in this paper. Rather than treating mathematics as background knowledge that students are expected to transfer independently, the instructional design emphasizes making mathematical reasoning visible, explicit, and instructional. Lessons and assignments are intentionally structured to help students recognize how mathematical ideas underpin programming constructs and to support the transfer of reasoning skills from mathematics to code.

This practice-based instructional context frames the design principles and lesson strategies discussed in the following sections and provides the foundation for the reflective observations presented later in the paper.

Design Principles for Integrating Mathematical Reasoning

The instructional strategies described in this paper are guided by a set of core design principles aimed at strengthening the connection between mathematical reasoning and programming practice in introductory computer science.

First, mathematical thinking is made explicit rather than assumed. Instruction deliberately surfaces the mathematical definitions, rules, and structures that underlie common programming constructs, such as integer division, remainder arithmetic, logical conditions, and operator precedence. By explicitly connecting Java operations to their mathematical foundations, students are guided to reason about why a construct behaves as it does, rather than memorizing language-specific behavior.

Second, instruction follows a progressive structure that moves from real-world problem contexts to mathematical representations and then to executable programs. Students are guided to focus on identifying relevant problem elements and relationships without immediately engaging in implementation details. These representations provide a clear analytical foundation that informs subsequent program design.

Third, instruction emphasizes reasoning before syntax by directly connecting mathematical models to programming constructs in Java. Students are guided to map mathematical relationships to variables, operators, program state, and control structures, and to reason about how sequential updates and conditional logic realize the intended behavior. This emphasis encourages students to view code as an implementation of established reasoning rather than a starting point for trial-and-error programming.

Finally, lessons, assignments, and learning outcomes are intentionally aligned so that programming tasks reinforce the reasoning processes emphasized during instruction. Assignments require students to articulate their reasoning, trace program state, and justify implementation choices, ensuring coherence between instructional goals, classroom activities, and assessment. Together, these principles form the conceptual foundation of the instructional approach presented in this paper.

The following sections illustrate how these principles are enacted through specific lesson structures and assignment designs. Rather than presenting classroom practices as isolated activities, each example demonstrates how mathematical reasoning is made explicit, progressively developed, and systematically connected to programming practice.

Lesson Design Strategy 1 (Principle 1): Making Mathematical Thinking Explicit

A central component of the first instructional strategy is to make mathematical reasoning explicit rather than assumed. Instruction focuses on revealing the mathematical definitions, rules, and structures that underlie common programming constructs, allowing students to understand *why* Java behaves as it does instead of relying on memorization or trial-and-error.

Confusion Surrounding Integer Division in Java

One effective way to make mathematical reasoning explicit is to examine how integer and real-number division in Java differ from mathematical division. Many students mistakenly assume that division behaves the same way in all contexts, which frequently results in errors when they transition from mathematics to programming.

When real numbers are used, division in Java behaves as expected mathematically:

```
double x = 9;
double y = 2;
double z = x / y;  // z = 4.5
```

In this case, the result preserves the decimal value, matching students' prior mathematical experience.

However, when both operands are integers, Java performs integer division. This means that the result must also be an integer. Because integers cannot store decimal values, Java discards the fractional part of the result:

```
int x = 9;
```

```
int y = 2;
int z = x / y;  // z = 4
```

The result is 4, not 4.5, and the value is not rounded. Instead, the fractional part is simply removed. This behavior often surprises students and leads to misconceptions about how division operates in Java.

This behavior is closely related to the mathematical concept of truncation**,** which differs from rounding. In mathematics, truncation refers to the process of removing the fractional part of a number without rounding, leaving only the integer portion unchanged. By explicitly connecting integer division in Java to this familiar mathematical concept, instruction helps students understand why the operation behaves as it does.

In this example, the result is not 4.5, and it is not rounded; rather, the decimal part is simply removed. When students grasp this underlying mathematical reasoning, they are less likely to make errors involving division in their programs and are better equipped to predict program behavior.

Connecting Integer Division and the Modulo Operator Through Quotient–Remainder Arithmetic

Another important opportunity to make mathematical thinking explicit arises when introducing the modulo (remainder) operator in Java. Students are often taught integer division and the modulo operator as separate concepts, which can obscure their shared mathematical foundation. In mathematics, integer division is formally defined by the quotient–remainder relationship**:** for integers a and b (with $b \neq 0$), there exist unique integers q and r such that

$$a = bq + r, \text{ where } 0 \leq r < |\, b \,|.$$

Java's integer division and modulo operators directly implement this mathematical definition. The division operator (`/`) computes the quotient, while the modulo operator (`%`) computes the remainder.

For example:

```
int q = 7 / 2;  // q = 3
int r = 7 % 2;  // r = 1
```

Here, the result of integer division is 3, and the remainder is 1, reflecting the mathematical identity $7 = 2 \times 3 + 1$. This example makes explicit that the fractional part discarded during integer division does not disappear arbitrarily; instead, it is captured by the remainder. Understanding this relationship helps students form a more accurate mental model of how integer arithmetic operates in Java. Rather than viewing `/` and `%` as unrelated operators, students learn to see them as complementary components of a single mathematical process.
Explicitly connecting the modulo operator to its mathematical meaning also clarifies its common uses in programming. For instance, checking whether a number is even or odd corresponds to testing whether the remainder after division by 2 is zero:

```
if (x % 2 == 0) {
    // x is even
}
```

By grounding the modulo operator in quotient–remainder arithmetic rather than treating it as a syntactic feature of the language, instruction reinforces the idea that Java's operators are precise computational realizations of well-defined mathematical concepts. This explicit connection reduces misconceptions, such as interpreting `%` as a percentage operator, and enables students to reason more confidently about integer arithmetic in their programs.

Connecting Piecewise Functions to Logical Conditions

Another common source of confusion arises from misunderstandings of logical conditions, particularly when students attempt to translate piecewise functions from mathematics into code. For example, the absolute value function is defined in mathematics as a piecewise function:

$$| x | = \begin{cases} -x & \texttt{if } x < 0 \\ x & \texttt{if } x \geq 0 \end{cases}$$

This definition specifies that different expressions apply under different conditions, and that exactly one case applies for any given input.

In programming, this same structure is implemented using logical conditions and conditional statements. The mathematical conditions ($x < 0$ and $x \geq 0$) become Boolean expressions that guide the program's execution:

```
if (x < 0) {
    absVal = -x;
} else {
    absVal = x;
}
```

Here, the `if-else` statement directly mirrors the structure of the piecewise function. The logical condition determines which computation is performed, ensuring that only one branch of code executes, just as only one case of the piecewise function applies.

By explicitly connecting piecewise functions to logical conditions in code, students can better understand that conditional statements are not arbitrary programming constructs, but rather computational realizations of mathematical definitions. This connection helps novice programmers feel more familiar with conditional logic and reduces the sense that programming syntax is unfamiliar or disconnected from prior mathematical knowledge. As a result, students can reason more effectively about program logic and are less likely to make errors when implementing condition-based behavior.

It is important to note, however, that this direct correspondence does not apply in all situations. A piecewise function assumes that its conditions are mutually exclusive and collectively exhaustive. If conditions overlap, fail to cover all possible inputs, or are intended to be evaluated

independently, a simple `if-else` structure no longer accurately represents the underlying logic. Making this limitation explicit helps students understand not only how to translate mathematical definitions into code, but also when such a translation is appropriate.

Errors in Applying the Order of Operations

Another common source of student error arises from misunderstandings of the order of operations when evaluating expressions in code. Students often assume that expressions in programming are evaluated strictly from left to right or rely solely on memorized rules such as PEMDAS, without understanding how these rules are applied in a programming context.
In mathematics, PEMDAS (Parentheses, Exponents, Multiplication and Division, Addition and Subtraction) provides a shorthand for the precedence of basic arithmetic operations. Java follows the same precedence rules for arithmetic operators, meaning that multiplication and division are evaluated before addition and subtraction.

For example:

```
int result = 2 + 4 * 3;  // result = 14
```

Here, the multiplication is performed before the addition, consistent with mathematical conventions.

However, PEMDAS is an incomplete model for understanding expression evaluation in Java. While it captures the precedence of basic arithmetic operations, it does not explicitly account for associativity, nor does it include unary operators, comparison operators, logical operators, or assignment, all of which play an important role in Java expression evaluation.

When operators share the same precedence level, Java evaluates them according to associativity, which for most arithmetic operators is left to right:

```
int result = 30 / 5 * 2;  // result = 12
```

Although this rule is also assumed in standard arithmetic, it is often left implicit in mathematics instruction and therefore overlooked by students. Making this evaluation order explicit helps students understand that Java does not change arithmetic rules but rather formalizes them in a precise and unambiguous way.

Java also applies unary operators before multiplication and division. Unary operators operate on a single operand and include operations such as unary negation (`-`), increment (`++`), and decrement (`--`):

```
int result = -4 * 2;  // result = -8
```

In addition to arithmetic operators, Java includes comparison operators (such as <, >, <=, >=, ==, and `!=`) and logical operators (such as `&&`, `||`, and `!`) that are not represented in PEMDAS. These operators are commonly used in conditional expressions and control-flow statements and depend on the results of arithmetic expressions.

Finally, assignment has one of the lowest precedence levels in Java. This means that the entire expression on the right-hand side of an assignment is fully evaluated before the result is stored in the variable on the left-hand side:

```
int a;
a = 4 + 5 * 2;
```

Overall, errors in applying the order of operations often arise not from a lack of mathematical knowledge, but from an incomplete understanding of how mathematical rules are formalized in programming languages. By explicitly highlighting these distinctions and making the evaluation process transparent, instructors can help students develop accurate mental models of expression evaluation.

Difficulty Constructing Correct Conditional Expressions

Another common challenge students face is constructing correct conditional expressions. Although students may understand individual comparison or logical operators in isolation, they often struggle to combine them into well-formed Boolean conditions.

In mathematics, compound conditions are often written compactly using inequalities, such as $0 \leq x < 6$. While intuitive in mathematics, this notation cannot be used directly in most programming languages.

In Java, each comparison must be stated explicitly and combined using logical operators:

```
if (x >= 0 && x < 6) {
    // statements
}
```

Students also frequently confuse the logical operators `&&` (AND) and `||` (OR), particularly when translating natural-language conditions:

```
if (x < 0 || x > 6) {
    // statements
}
```

By explicitly connecting mathematical inequalities, logical reasoning, and Java's Boolean expression syntax, instruction helps students reduce common logical errors and strengthens their ability to reason about program control flow.

Lesson Design Strategy 2 (Principle 2): Translating Real-World Problems into Mathematical Expressions

While Strategy 1 focuses on making the mathematical rules underlying programming constructs explicit, Strategy 2 emphasizes translating real-world problem scenarios into mathematical representations before any code is written. The goal of this strategy is to help students recognize

that programming problems are structured reasoning tasks based on quantities, relationships, and constraints.

Lessons using this strategy typically begin with contextualized problem statements rather than code. Students are first asked to interpret the problem in natural language, identify relevant quantities, and describe how those quantities are related. Instruction then guides students through problem decomposition, such as naming variables, determining units, and expressing relationships using equations or inequalities. This focus on mathematical modeling helps students understand the problem structure before engaging with programming syntax.

By emphasizing the translation of real-world scenarios into mathematical expressions, students develop a clearer understanding of how programming tasks are grounded in formal reasoning rather than trial-and-error coding. Logical expressions are introduced alongside numerical equations to prepare students for conditional statements and decision making in later programming steps. This approach reinforces that arithmetic reasoning and logical reasoning work together in computational problem solving.

The following laboratory exercise illustrates how this design principle is enacted in practice:

```
Students are asked to write a program that takes a total amount
of change, given as an integer number of pennies, and outputs
the equivalent amount using the fewest number of coins. The
available coin types include Dollars, Quarters, Dimes, Nickels,
and Pennies, with correct handling of singular and plural coin
names (e.g., 1 Penny versus 2 Pennies).
```

Although the lab culminates in a complete Java program, the instructional sequence begins with structured reasoning and mathematical modeling before students proceed to implementation.

Before writing any code, students are guided to construct a mathematical model of the problem. The total amount of change is first represented entirely in pennies, establishing a consistent unit of measurement. Each coin type is then associated with a fixed numerical value:

- 1 Dollar = 100 pennies
- 1 Quarter = 25 pennies
- 1 Dime = 10 pennies
- 1 Nickel = 5 pennies

Students are asked to explain why starting with the largest denomination minimizes the total number of coins, reinforcing the idea that algorithmic decisions should be justified mathematically. The task is then framed as a repeated application of integer division and remainder operations. For a given remaining amount R:

- The number of dollars is computed as $R \div 100$
- The remaining amount is updated as $R \ mod \ 100$

This quotient–remainder reasoning is repeated for each denomination:

- Quarters: $R \div 25$, remainder $R \ mod \ 25$
- Dimes: $R \div 10,$ remainder $R \ mod \ 10$
- Nickels: $R \div 5,$ remainder $R \ mod \ 5$
- Pennies: the final remaining amount

At this stage, students are encouraged to articulate the reasoning verbally or in writing, explaining how each step reduces the remaining amount and why the remainder must be carried forward. This emphasis helps students understand that the mathematical model determines the correctness of the solution, independent of how it is later implemented in code.

Only after the mathematical structure is fully established are logical considerations introduced. Handling singular and plural coin names is presented as a separate logical decision problem, reinforcing the distinction between numerical computation and conditional reasoning. This separation helps students recognize that complex programming tasks often consist of multiple layers of reasoning, each of which can be addressed systematically.

This example demonstrates how translating a real-world scenario into mathematical expressions provides a clear roadmap for program design. Students who engage in this modeling process are better able to reason about correctness, anticipate edge cases, and avoid trial-and-error coding. More importantly, they begin to see programming as the execution of a well-defined mathematical plan rather than as an exercise in syntax manipulation. This perspective supports deeper conceptual understanding and prepares students for more complex algorithmic reasoning in later coursework.

Lesson Design Strategy 3 (Principle 3): Connecting Mathematical Concepts to Programming Constructs

While Strategy 2 emphasizes translating a real-world problem into mathematical expressions, Strategy 3 focuses on helping students connect those expressions to concrete programming constructs in Java. The goal of this strategy is to help students see code as an implementation of prior reasoning rather than a starting point for trial-and-error programming. The same change-making task is revisited, but the focus shifts from mathematical modeling to program structure, variable selection, operator use, and the organization of output.

Mapping Mathematical Reasoning to Program State and Operators

Students begin with the quotient–remainder model developed earlier and map each part of that model to elements in the program. The remaining amount of change, denoted as R in the mathematical model, is represented by the variable `remaining`. Each coin count (dollars, quarters, dimes, nickels, and pennies) is stored in a separate integer variable. Integer division is used to compute the number of coins, and the modulo operator is used to update the remaining amount.

This explicit mapping helps students understand that Java's arithmetic operators directly implement the mathematical reasoning they have already established. Instruction also emphasizes that the algorithm proceeds step by step. Each calculation updates the program state, and later results depend on earlier updates. Students are asked to explain why the value of `remaining` must be updated after each denomination and how this update reflects the remainder in the quotient–remainder relationship.

Connecting Logical Reasoning to Conditional Statements

After implementing the arithmetic portion of the program, students address the output requirements. Handling singular and plural coin names is introduced as a logical decision problem. The program must select the appropriate label based on the computed coin count, which provides a natural context for conditional statements and Boolean expressions such as `count == 1`.
Students are encouraged to distinguish between numerical computation and output formatting. This distinction reinforces the idea that correct programs require both arithmetic reasoning and logical reasoning, even when the underlying mathematical model is correct.

Java Implementation

```
import java.util.Scanner;

public class CoinCalculator {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Input: total change amount in pennies
        System.out.print("Enter the total amount in pennies: ");
        int totalPennies = input.nextInt();
        int remaining = totalPennies;

        // Mathematical model: quotient-remainder for each denomination
        int dollars = remaining / 100;
        remaining = remaining % 100;

        int quarters = remaining / 25;
        remaining = remaining % 25;

        int dimes = remaining / 10;
        remaining = remaining % 10;

        int nickels = remaining / 5;
        remaining = remaining % 5;

        int pennies = remaining;

        // Output: one coin type per line, with correct singular/plural naming
        if (dollars > 0) {
            System.out.println(dollars + (dollars == 1 ? " Dollar" : " Dollars"));
        }
        if (quarters > 0) {
            System.out.println(quarters + (quarters == 1 ? " Quarter" : " Quarters"));
        }
        if (dimes > 0) {
            System.out.println(dimes + (dimes == 1 ? " Dime" : " Dimes"));
        }
        if (nickels > 0) {
            System.out.println(nickels + (nickels == 1 ? " Nickel" : " Nickels"));
        }
        if (pennies > 0) {
```

```
            System.out.println(pennies + (pennies == 1 ? " Penny" : " Pennies"));
        }

        input.close();
    }
}
```

Instructional Emphasis During Implementation

During implementation, students are asked to explain the purpose of each variable and the reasoning behind each line of code. For example, students explain that `dollars = remaining / 100` computes the quotient in the expression R ÷ 100, and that `remaining = remaining % 100` updates the remainder. This emphasis on explanation reinforces the principle of reasoning before syntax and discourages copy-and-paste coding.

Finally, students test their programs with sample inputs and trace how the value of `remaining` changes after each step. This tracing activity helps students connect mathematical reasoning to program execution and encourages them to view debugging as a reasoning process rather than a guessing exercise.

By explicitly connecting mathematical concepts such as quotient–remainder structure and logical decision rules to programming constructs including variables, operators, sequential updates, and conditional statements, Strategy 3 helps students develop stable mental models of program behavior. As a result, students are better able to write correct programs, identify errors, and approach problem solving in a systematic and confident way.

Observations and Reflections

The observations presented in this section are grounded in the author's reflective teaching practice across multiple semesters of instruction in introductory computer science. Drawing on sustained classroom experience, several consistent patterns were observed in student work and learning behaviors. Students who engaged more explicitly with mathematical reasoning demonstrated improved ability to explain their code, articulate underlying logic, and identify errors through systematic reasoning rather than trial-and-error approaches. Informal assessments, including programming assignments and code explanations submitted by students, showed similar patterns of improvement in students' ability to reason about program behavior and justify implementation choices. Many students also reported increased confidence in programming tasks as they began to recognize familiar reasoning processes from mathematics.

Over time, students' problem-solving approaches shifted from syntax-driven experimentation toward more deliberate analysis and planning. Rather than immediately attempting to code, students increasingly focused on understanding problem structure, anticipating outcomes, and reasoning through solution strategies prior to implementation. These observations suggest positive instructional trends; however, they are derived from reflective classroom experience and informal classroom assessments rather than controlled experimental study. As such, they should be interpreted as qualitative, practice-based insights rather than as generalizable empirical findings.

Several instructional lessons emerged from this experience. Explicitly integrating mathematical reasoning supported deeper conceptual understanding for many students, but it also required careful pacing and intentional scaffolding. Some students initially resisted the emphasis on reasoning, expressing a preference for focusing on syntax and immediate code execution, which necessitated ongoing instructional adjustment.

Common challenges included introducing mathematical depth too quickly or assuming familiarity with symbolic reasoning. Maintaining an appropriate balance between mathematical rigor and programming scope proved essential for sustaining student engagement. These considerations may be particularly relevant for instructors seeking to adopt similar approaches in introductory programming courses.

Conclusion

This paper presented a reflective examination of instructional strategies designed to bridge mathematical reasoning and programming practice in an introductory computer science course. Grounded in the author's sustained classroom experience and informed by prior research in computer science education, the approach treats mathematical reasoning as an explicit instructional objective rather than as assumed background knowledge.

Across the three strategies presented, mathematical thinking is progressively foregrounded, structured, and operationalized in instructional practice. Strategy 1 makes the mathematical foundations of programming constructs explicit by connecting Java operations to familiar mathematical definitions and rules. Strategy 2 emphasizes the translation of real-world problem scenarios into mathematical representations, supporting students in developing structured problem-solving models before engaging with code. Strategy 3 then connects these mathematical models to concrete programming constructs, reinforcing the view of code as a precise computational realization of prior reasoning rather than a starting point for trial-and-error experimentation.

Although this work does not claim causal impact or generalizable outcomes, it offers practice-based insights into how deliberate lesson and assignment design can help students develop more accurate mental models of program behavior and reduce common misconceptions in introductory programming. The strategies described are not specific to Java and may be adapted to other programming languages and instructional contexts. Examples of student solutions and classroom assessments indicate that these strategies help students better articulate the reasoning underlying their programs. More broadly, this work highlights the value of closer alignment between mathematical reasoning and computer science instruction, particularly in early coursework. Future work may involve more systematic assessment of learning outcomes and further exploration of how such approaches can be integrated across curricula to support students transitioning into computer science.

References

[1] Graham, R. L. (1994). *Concrete mathematics: a foundation for computer science*. Pearson Education India.

[2] Association for Computing Machinery and IEEE Computer Society, *Computer Science Curricula 2023 (CS2023), Version Beta*. ACM/IEEE-CS, 2023.

[3] Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education*, *13*(2), 137-172.

[4] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., ... & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*(4), 119-150.

[5] Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33-35.

[6] Medeiros, R. P., Ramalho, G. L., & Falcão, T. P. (2018). A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, *62*(2), 77-90.

[7] Scherer, R., Siddiq, F., & Viveros, B. S. (2020). A meta-analysis of teaching and learning computer programming: Effective instructional approaches and conditions. *Computers in Human Behavior*, *109*, 106349.

[8] Wonnacott, D. G., & Osera, P. M. (2019). A bridge anchored on both sides: formal deduction in introductory CS, and code proofs in discrete math. *arXiv preprint arXiv:1907.04134*.

[9] Singh, S. (2025). *Addressing Challenges in Computer Programming Education: An Action Research Approach for Effective Student Support* (Doctoral dissertation, Northeastern University).

[10] Hattie, J., & Yates, G. C. (2013). *Visible learning and the science of how we learn*. Routledge.

[11] Bruner, J. S. (1960). The Process of Education, Harvard, Univ. *Press, Cambridge, Mass*.

[12] Petre, M. (1995). Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, *38*(6), 33-44.

[13] Biggs, J., & Tang, C. (2003). Teaching for quality learning. *Buckingham: Society for*.