PEDAGOGY · TECHNICAL EDUCATION · PROGRAMMING INSTRUCTION

Why Before How.

Most programming instruction starts with syntax — the visible, demonstrable surface of a language. The PARTS method starts somewhere earlier, and it changes everything.

By Avalynn Circe · Author, JavaScript: The Parts · AvalynnCirce.com

---

There is a particular moment that anyone who has taught programming will recognize. A student has followed all the instructions. They have completed the exercises. They can reproduce the examples. And then they encounter something slightly different from what they have seen — a variation, a combination, a context the tutorial did not cover — and they stop completely. They have learned to pattern-match. They have not learned to think.

This gap between demonstrated competence and actual understanding is not a failure of individual students. It is, in large part, a predictable consequence of how technical subjects are typically taught: feature-first, syntax-first, demonstration-first. The approach prioritizes getting students to a visible result quickly, which serves engagement metrics and tutorial completion rates. What it often fails to build is the deeper conceptual scaffolding that makes a learner genuinely capable of applying knowledge to new problems.

The PARTS method is a structured pedagogical framework developed to address this gap specifically in the context of programming instruction. It is not a learning theory in the abstract — it is a sequencing discipline, a protocol for the order in which concepts are introduced and the questions each introduction must answer before moving forward. It emerged from direct observation of where learners lose the thread, and from a conviction that the question most teaching skips is the one that matters most.

That question is: why does this exist at all?

The Problem With Feature-First Teaching

The dominant model for teaching programming languages is organized around features. Here is a variable. Here is a function. Here is a loop. Here is an array. Each feature is demonstrated, exercised, and then the learner moves on to the next.

This approach has real virtues. It is easy to sequence. It maps naturally to language documentation. It produces working code quickly. And for learners who already have strong intuitions about computational thinking — who have, in other words, already internalized the mental model of programming through prior experience — it works well enough. The demonstrations slot into a framework they already possess.

The difficulty is the quiet assumption embedded in the model: that the framework will form on its own if the learner sees enough examples. For a significant portion of learners, it does not. Concepts arrive without context. Syntax appears before meaning.

The cumulative effect is a learner who can produce code by imitation but cannot generate code from reasoning. JavaScript — or any language taught this way — becomes something to survive rather than something to understand.

Feature-first teaching also tends to introduce vocabulary carelessly. A variable is called a 'container.' A function is called a 'reusable block of code.' These metaphors are approximate. They communicate something. But they also plant assumptions that become obstacles later. The learner who understands a variable as a container will be confused by the behavior of references, will struggle to understand closures, and will write bugs that they cannot explain because their mental model of the underlying mechanism is wrong.

> ***When a student hears a new word and forms an incorrect assumption about its meaning, that assumption is sticky. It persists long after they can write working code. PARTS is designed to prevent the assumption from forming in the first place.***

The PARTS Framework

PARTS is an acronym representing a five-step sequence applied consistently to the introduction of each new concept:

Step	Question It Answers	Teaching Function
P — Problem	Why does this exist?	Establishes motivation before introducing the concept. Students understand the need before they encounter the solution.
A — Answer	What solves that need?	Introduces the concept as a response to a real problem, not as an arbitrary language feature.
R — Reasoning	Why does this answer make sense?	Builds intuition for the concept's shape and constraints. Prevents the 'I can use it but I don't understand it' gap.
T — Terms	What words make this precise?	Establishes exact vocabulary before syntax. Corrects premature mental models and sticky metaphors.
S — Syntax	How is this written in code?	Introduces notation only after the idea is understood. Syntax becomes a representation of a known concept, not the concept itself.

The order is not arbitrary. It reflects a theory of how conceptual understanding is actually built — and how it differs from the procedural knowledge that demonstrations alone can produce.

Problem comes first because motivation is the precondition for understanding. A learner who does not feel the need that a concept addresses has no anchor for the concept when it arrives. They can memorize it, but they cannot reason with it. When a student understands that closures exist because functions need a way to preserve access to their surrounding scope after that scope has been popped from the call stack, closures stop being a piece of arcane syntax to be memorized and become a logical consequence of how the language was designed to work. The concept is earned.

Answer follows from Problem precisely because it has been set up. The student is not being handed a tool without context. They are being shown how one specific problem was solved.

Reasoning is the step most teaching omits entirely. It is not enough to show that something works. A learner who understands why the design of a concept is what it is — why closures work by capturing references rather than values, why array methods return new arrays rather than modifying the original — has a mental model robust enough to handle novel situations. Reasoning builds intuition. And intuition is what separates a learner who can follow instructions from one who can solve problems.

Terms come before Syntax deliberately. This is the most distinctive structural choice in the PARTS method, and it is worth examining carefully.

The Vocabulary Problem

In most technical instruction, terminology arrives simultaneously with syntax or after it. The student sees the code first, learns what it does, and then acquires the word for it. This sequence seems natural — show the thing, then name it. But it has a significant drawback.

When students encounter unfamiliar terminology without prior definition, they do not suspend judgment and wait for clarification. They form assumptions. They match the new word to the nearest familiar concept, or they infer meaning from context, or they pattern-match to something they have seen before. These assumptions are, as any experienced educator knows, remarkably persistent. Correcting a wrong mental model is substantially harder than building a correct one from the start — because the wrong model has already been connected to behavior, validated by working examples, and integrated into how the student thinks about related concepts.

A student who believes a variable is a 'container' — a box that holds a value — will have that model reinforced every time a simple assignment works as expected. It will not be challenged until they encounter a situation where the container metaphor breaks down: reference behavior, closures, the distinction between a binding and a value. By that point, the incorrect model is deeply embedded. Dislodging it requires not just introducing the correct model but actively dismantling the wrong one, which creates confusion and erodes confidence.

PARTS addresses this by treating conceptual clarity as a prerequisite for notation. The Terms step establishes precise vocabulary — binding rather than container, invocation

rather than run, parameter rather than placeholder — before any code is written. When the student then encounters the Syntax, they are not looking at unfamiliar symbols trying to guess what they mean. They are reading a notation for something they already understand.

> ***Syntax is the last step because syntax is the notation for an idea — and you cannot read a notation for something you have not yet understood. PARTS gets the idea right first.***

Syntax as Notation, Not Foundation

This reframing of syntax — from the entry point of instruction to its culmination — is the philosophical center of the PARTS method. It reflects a position about the nature of programming languages that is easy to state but consequential in its implications.

Syntax is not the language. Syntax is the language's notation. The language is the system of concepts, constraints, and relationships that the notation represents. You can learn to write JavaScript syntax without understanding JavaScript. You can produce code that works without understanding why it works. But you cannot extend, debug, or transfer that knowledge to new problems without the underlying conceptual model. Syntax-first instruction builds the notation without building what the notation is notation for.

To use an analogy from another domain: a student can learn to read musical notation — to map symbols to key positions — without understanding harmony, rhythm, or musical structure. They can play what is written in front of them. What they cannot do is improvise, compose, or diagnose why a passage sounds wrong. The notation was learned without the music. PARTS insists on learning the music first.

In practice, the Syntax step in PARTS is often the shortest. Once the concept is understood and the vocabulary is precise, the notation is usually straightforward. The work has already been done. The symbols are just a compact way of writing what the student already knows.

Sequencing as Argument

The PARTS method operates at two levels simultaneously. Within each concept, it governs the order of introduction: Problem, Answer, Reasoning, Terms, Syntax. But it also shapes the macro-level sequencing of concepts through the course as a whole.

In PARTS-structured instruction, concepts are not introduced in the order they appear in language documentation or in the order they happen to come up in common tutorials. They are introduced in the order in which each one creates the need for the next. Values come first because a language must first establish what kinds of meaning it can hold. Variables come second because once values exist, the problem is how to refer back to them. Expressions come third because once values can be referred to, the question is how to combine and compare them. Conditions come fourth because once expressions can produce true-or-false results, the question is how to choose between paths.

Each concept earns its place in the sequence by solving a problem the previous concept left open. The curriculum is not a list of topics — it is an argument. By the time the student encounters closures or promises or the event loop, they have arrived at those concepts through a chain of reasoning, each step of which they understood before taking the next. The advanced topic is no longer mysterious. It is the answer to a question the student was already asking.

Observed Effects in Practice

The PARTS method was developed in the context of JavaScript: The Parts, a technical reference book aimed at working developers seeking a deeper understanding of the language. The feedback from readers has been instructive, and it consistently points to the same phenomenon: learners who already know JavaScript encounter PARTS-structured explanations and find themselves understanding, for the first time, why behaviors they have been working around for years are shaped the way they are.

This is significant. The learners in question can already write JavaScript. They are not beginners. But their knowledge was built on a syntax-first foundation, and certain concepts — the specific mechanics of closures, the execution model of the event loop, the distinction between primitives and references — were understood procedurally rather than conceptually. They knew what to do. They did not know why. PARTS-structured treatment of those concepts, even for experienced practitioners, produces recognition rather than confusion: yes, that is why it works that way.

The implication for classroom instruction is that the benefit of PARTS is not limited to beginners. It is relevant wherever a learner's working knowledge exceeds their conceptual understanding — which describes a substantial portion of self-taught and tutorial-educated programmers.

> ***The goal of PARTS is not to produce learners who can demonstrate the language. It is to produce learners who understand why the language is shaped the way it is — and who can therefore reason about it in situations no tutorial has covered.***

Relationship to Established Learning Theory

The PARTS method was developed from direct pedagogical observation rather than derived from learning theory, but its structure is consistent with several well-established frameworks in educational research.

The Problem-first orientation reflects the constructivist tradition associated with Piaget and elaborated by later researchers in situated learning: meaningful learning occurs when new concepts are connected to problems the learner already perceives as real. Information encountered without a felt need attaches to memory weakly, if at all. PARTS operationalizes this by making the problem statement the mandatory first step

— not a motivational preamble that instructors can skip, but the structural prerequisite for everything that follows.

The Terms-before-Syntax sequence reflects research on conceptual change learning, particularly the work on misconception correction in STEM education. Studies across mathematics, physics, and computer science consistently show that incorrect prior conceptions, once formed, resist correction and interfere with new learning. Proactive vocabulary instruction — establishing precise definitions before the learner can form approximate ones — is more effective than reactive correction. PARTS builds this into the sequence by design.

The macro-level concept sequencing — where each concept is introduced as the answer to a problem the previous concept created — reflects principles associated with spiral curriculum design and with what cognitive scientists call elaborative interrogation: the practice of building understanding by continuously asking why, not just what. The PARTS curriculum does not present a collection of facts. It presents a developing argument. Each chapter advances the argument by one step.

Limitations and Scope

The PARTS method is a sequencing discipline, not a complete pedagogical system. It specifies the order in which concepts should be introduced and the questions each introduction must answer. It does not specify assessment design, practice structure, project sequencing, or the management of a learning community. These are separate concerns, and PARTS is compatible with a range of approaches to each of them.

PARTS is also most naturally applicable to conceptually dense technical subjects — programming languages, formal systems, mathematical structures — where the gap between notation and concept is significant and where incorrect mental models have compounding effects on later learning. Its applicability to domains where surface skill and underlying concept are more closely aligned is an open question.

The framework has been applied primarily in the context of JavaScript instruction. Extension to other programming languages, to other technical disciplines, or to non-technical subjects would require validation that the structural approach transfers — that the problem-first, syntax-last ordering produces similar benefits in different conceptual terrains.

Conclusion

The problem PARTS addresses is not a new one. Educators have observed the gap between procedural fluency and conceptual understanding in technical subjects for as long as technical subjects have been formally taught. What PARTS offers is a specific, implementable structural response to that problem — a protocol that any instructor can apply to any concept in any technically dense subject.

Its central claim is modest but demanding: that learners understand things better when they first understand why those things need to exist. That vocabulary should be established before notation. That reasoning should be made explicit rather than left to emerge on its own. And that a curriculum organized as an argument — where each

concept earns its place by solving the problem the previous concept left open — produces learners capable of generating knowledge, not just reproducing it.

The goal is not to produce learners who can demonstrate the language under familiar conditions. It is to produce learners who understand the language well enough to reason about it in conditions they have never seen — which is, ultimately, the only condition that matters in practice.

About the Author

Avalynn Circe is a software developer, business analyst, and technical writer based in St. Paul, MN. She developed the PARTS method while writing JavaScript: The Parts, a 378-page technical reference for working JavaScript developers published on Leanpub. The method emerged from sustained observation of where learners lose conceptual grounding in programming instruction and represents her attempt to build a teaching framework that addresses the root cause rather than its symptoms.

JavaScript: The Parts is available at leanpub.com/jsparts. Portfolio and contact: AvalynnCirce.com