JavaScript: The Parts

---

and How the Parts Fit Together

Table of Contents

INTRODUCTION

I needed this book

My path into JavaScript did not begin with a computer science degree or a formal sequence of classes. Like many people, I learned it on my own, in fragments, one problem at a time. I could follow tutorials, copy patterns, and get code running. Eventually I could build real projects.

But something was missing.

I knew how to repeat what I had seen. I did not always know why it worked. And when I did not understand why something worked, I also could not understand why it failed.

That gap matters more than it first appears.

You can make progress for a long time by imitation. You can memorize enough syntax, enough patterns, enough habits to stay productive. For a while, that can feel like understanding.

Eventually the cracks show. Something breaks in a way no tutorial covered. A familiar pattern almost works, but not quite. You sit down to explain your own code and realize you cannot.

One of the most common moments this happens is a job interview. Someone asks you to explain how JavaScript handles variables. You have written code that does exactly that, dozens of times. But explaining it is a different thing entirely. The words are not there. The understanding that would produce those words is not there either.

That is not a gap in your confidence. It is a gap in your foundation.

I Was an Imposter

I wrote this book because I needed it.

For a long time I felt like I was faking it. I could build things. I could get code to run. But I could not always explain what the code was doing, or why it worked, or what would happen if something changed. I had the output without the understanding. And I knew it.

What I needed was a way into JavaScript that started earlier than syntax. Not a bag of features to memorize, but a way of seeing how the language fits together — and why each part of it had to exist in the first place.

I discovered what I had been missing when I started making study flashcards. I wrote down concepts, found the most accurate definitions I could, and organized the cards so I could see how the ideas connected. That process changed everything. JavaScript stopped feeling like a pile of disconnected terms and started feeling like a structure.

This book grew out of that process.

Imposter Syndrome Has a Cause

If that experience sounds familiar, you are probably not alone.

Many people who code — even people who code well — carry a quiet feeling that they do not really belong. That they are faking it. That sooner or later someone will ask them a question they cannot answer and the whole thing will fall apart.

That feeling has a name: imposter syndrome. And while it is easy to treat it as a confidence problem, it is usually something more specific than that.

If you learned JavaScript by imitation — following tutorials, copying patterns, getting things to work without always knowing why — then there are real gaps in your

understanding. Not because you are not smart enough, and not because you did not work hard enough, but because the way the material was taught left those gaps open.

Feeling like an imposter is a reasonable response to genuinely not knowing something you feel you should know. Confidence tends to follow understanding. Learn the thing, and the feeling takes care of itself.

That is what this book is for.

The PARTS Method

Each concept in this book is introduced the same way.

Problem — What need gives rise to this idea?
Answer — What concept meets that need?
Reasoning — Why does that answer make sense?
Terms — What words make the idea precise?
Syntax — How is that idea written in JavaScript?

That order matters.

Most people encounter syntax first because syntax is the visible part — the thing you can point to on a page, the thing a tutorial can demonstrate in thirty seconds. But visible does not mean foundational.

To build a language, you do not start with symbols. You start with a need.

A language first has to establish what counts as a value — what kinds of meaning the language can hold at all. Once values exist, the next problem is how to refer back to them. That is where variables come from. Once values can be referred to, the next problem is how to combine and compare them. That is where expressions come from. Once expressions can produce true-or-false results, the next problem is how to choose between different paths. That is where conditions come from.

Each idea earns its place by solving a problem the previous idea could not yet solve.

That is the spirit of this book: not just here is how to use it, but here is why it had to exist at all.

These five steps shape every chapter. You will not always see them labeled — the chapters are written to feel like continuous reasoning, not a form being filled in. But the logic is always moving in the same direction: from the need, to the idea, to the reason, to the words, to the code.

The Machine

Most people approach JavaScript like it has blurry edges.

Like there are hidden places where things happen that cannot be predicted or explained. Like some code works and some code does not, and the difference is not always clear.

That feeling is understandable. But it is not accurate.

JavaScript is a machine.

A very precise, very literal, very mechanical machine. It does not guess. It does not approximate. It follows rules — the same rules, every time, without exception. This does not make it simple. But it makes it knowable.

Every behavior has a cause. Every output follows from an input. Every surprising result is only surprising until you understand the rule behind it. Once you see the rule, the surprise disappears. What looked like a cliff to fall off of turns out to be a step with a very clear edge.

That is what this book is about.

Not memorizing syntax. Understanding the machine.

When you understand the machine, you stop guessing and start reasoning. You stop fearing the unexpected and start asking: what rule produced this? You stop treating errors like accusations and start treating them like information.

Programs are mechanical in exactly this way. This responds to that. This triggers that. One thing leads to the next, in a sequence that can always be followed if you know where to look.

That kind of thinking is not intimidating. It is the opposite.

It is a puzzle. And puzzles are solvable.

A Note on AI

AI tools were used during the drafting process — for prototyping ideas, stress-testing explanations, and proofreading. The thinking, the structure, the voice, and the decisions about what to say and how to say it are my own. This book was not generated by AI. It was written by a person who needed it.

A Note on Language

This book uses language carefully.

Every definition has been chosen to be as precise as possible without becoming unnecessarily technical. That balance is harder than it sounds. Most introductory books simplify their definitions to make them easier to absorb — and in doing so, they quietly plant misconceptions that the reader has to unlearn later.

This book tries not to do that.

A variable is not described as a container that holds a value — because that image breaks down later. Reassignment is not described as changing a value — because the value does not change. The name does.

Those distinctions are not pedantic. They are the difference between an explanation that works now and one that keeps working as the ideas get more complex.

Where a simplified explanation would be genuinely harmless, this book uses it. Where a simplification would plant a misconception, this book chooses precision — and trusts the reader to handle it.

Who This Book Is For

This book is for anyone who has ever thought: I can make this work, but I do not really understand it.

It is for people who are self-taught, who learned in fragments, and who can already build things but still feel that their understanding has holes in it.

It is for people preparing for job interviews who want to be able to explain their code, not just write it.

And it is for people who have ever felt like an imposter. If that is you, this book is not going to tell you to believe in yourself more. It is going to give you the understanding that makes that feeling go away on its own.

You do not need to be a complete beginner. You do not need to be advanced. You only need to want more than survival knowledge — to want to think clearly about code, not just produce it.

Looking Forward

To understand JavaScript, it helps to start with the problem it was created to solve. That story starts with the browser.

ENTER JAVASCRIPT

---

a solution to a problem

Before JavaScript, Browsers Mainly Delivered Documents

In the early web, a browser's main job was to request a document from a server and display it on the user's screen. A browser could ask for a page, the server could respond with HTML, and the browser could render it.

That model made the web possible. It also set an early limit on what a page could be.

A page could contain text, images, links, and forms. Once it loaded, though, the page itself had very little ability to respond to what the user did next. Viewing a page was not much different from reading a newspaper. You could read it, follow links, and submit a form. But the page could not think. It could not react. It simply sat there.

The early web was closer to a digital publication than an interactive system.

The Problem: The Page Could Not Respond From Inside the Browser

This became a problem because users do not only read. They type, click, make choices, make mistakes, and change their minds.

A document can present information. An interactive system has to do more. It has to receive input, interpret it, and respond.

In the early web, most of that response logic lived on a distant server, not in the user's browser. When the user did something important — submitting a form, for example — the information had to travel across the network. The server would inspect it, decide what it meant, and send back a new page.

That made even small interactions feel clumsy. Consider a user filling out a form and leaving a required field empty. The usual flow was to submit the form, send the data across the network, wait for the server to inspect it, and then reload the page just to show an error message.

The user acts now. The page responds later. The feedback feels disconnected from the action that caused it.

The Missing Piece Was Programmable Logic in the Browser

The problem was not simply that pages were not interactive enough. More precisely, page authors lacked a way to work with data inside the browser after the page loaded.

They needed a way to receive user input, store temporary values, compare values, make decisions, react to events, and update the page — without sending every small step back to the server.

The browser already had built-in capabilities of its own. What web pages needed was a programmable layer that could inspect values and respond in real time.

In other words, the browser needed a scripting language for the page itself.

Enter JavaScript

JavaScript was created to provide that missing capability.

With JavaScript, a page could do more than sit there after loading. It could receive input, store values, perform comparisons, make decisions, respond to events, and update parts of the page directly in the browser — immediately, without waiting for the server.

Before JavaScript, most application logic lived on the server, and the browser mostly displayed the result. JavaScript changed that balance. Logic could now happen right where the user was. Checks could happen immediately. Updates could happen in place.

Responses became immediate rather than delayed. The page started to feel alive.

Looking Forward

JavaScript exists to let the browser work with data and respond to what users do. User actions create data. Typing creates input. Clicking creates information the browser can inspect. Form fields, button presses, and page events all become things JavaScript can work with.

But before JavaScript can store a value, compare it, or make a decision about it — before any of that is possible — something more basic has to be true.

The language has to know what a value is.

Now that you know why this book exists and why JavaScript exists, let's take a high-level view of your path ahead, the big picture.

THE BIG PICTURE

---

see the journey before the details

The Whole Journey, in One Sentence

We begin with simple values, follow the problems that give rise to new ideas, learn the terms for those ideas, see how they appear in syntax, and gradually build systems that can respond to input, requests, and failure.

Before we begin, it helps to step back and see where we are going.

You do not need to memorize anything in this section, and you do not need to understand it fully yet. Its purpose is to give you the shape of the path ahead. This book moves in a deliberate order. Each step solves a problem the earlier step could not yet solve, and each new idea grows out of the limits of the one before it.

It Starts With Something Simple

Everything begins with values.

A number. A piece of text. Something that is true or false.

These are the smallest units of meaning in the language. They do not do much on their own, but they are the raw material for everything else. Before JavaScript can have variables,

expressions, conditions, or functions, it has to solve a more basic problem: what counts as a value in the language at all?

Holding Onto Meaning

Without a name, a value has no way to be referred to after the moment it first appears.

Variables give a value a name — a way to refer back to it after the moment it first appeared. A program can return to it, compare it, use it in new expressions, and refer to it again whenever it is needed. This is one of the first moments where programming begins to feel useful, because a value no longer has to vanish the instant it appears.

Creating Relationships

Once values persist, they can be combined, compared, and related to one another.

This is where logic begins to take shape. The program is no longer just holding values. It is doing something with them. Expressions and operators let values interact, and those interactions begin producing results that matter.

Choosing What Happens

Programs become more powerful when they can respond differently depending on a situation.

If this is true, do this. Otherwise, do something else.

Conditions make that possible. A program can check what is true and choose what happens next. This is where code begins to branch instead of merely proceed.

Repeating the Check

Some situations do not end after one decision.

A program may need to keep counting, keep checking, or keep applying the same step until the situation changes. Loops make that possible by repeating a check and continuing only while that check still holds. This is how repetition becomes structure instead of copy-and-paste.

Reusing Logic

When the same pattern appears in more than one place, it becomes useful to name it and reuse it.

Functions turn repeated logic into tools. Instead of rewriting the same steps again and again, the program can define that logic once and call on it whenever needed. This is one of the points where code starts becoming easier to grow.

Visibility

As programs grow, it becomes important to control what can be seen and used from where.

Not every name should be visible everywhere. Scope creates boundaries around visibility, which helps different parts of a program stay clear and prevents accidental interference. This is what keeps growing code from collapsing into one shared blur.

Keeping Related Values Together

Real problems rarely stay small enough to fit inside single values.

Arrays let us keep many values together in one ordered place. Objects let us keep related values together as parts of one thing. This is where programming starts to model the kind of

data that actual problems involve — not just isolated numbers and pieces of text, but structured information with shape and meaning.

Modifying Groups of Values

Once values are gathered into collections, the next question becomes what we want to do with them.

A program may change each item, keep only some items, or combine many items into one result. This is where collections stop being storage alone and start becoming expressive tools.

Connecting Data and Logic

Logic does not always belong somewhere separate from the data it works with.

Methods allow logic to live on the objects it affects. At that point, data and logic begin to form small systems instead of remaining loose pieces. The program starts to feel more like a set of connected parts and less like a pile of separate operations.

Working Now and Later

Not everything happens right away.

Some work begins now and finishes later. A program has to continue functioning while a timer runs, a request travels, or a response is still on its way. This introduces a different way of thinking about execution — separating the moment something starts from the moment it finishes.

Inputs, Requests, and Errors

Programs respond to people and systems.

They respond to input, ask for things they do not already have, and deal with situations they do not fully control. This is where programming begins to feel like real application work, because the program must now listen, reach outward, and cope with uncertainty.

What This Book Is Really About

This book is not only about learning JavaScript.

It is about learning how to think in systems: how small ideas build into larger ones, how concepts connect, and how to figure things out without guessing. JavaScript is the material. Clear reasoning is the larger skill.

Once you can reason clearly about code, something changes. New problems stop feeling like walls. You may not know the answer right away — but you will know how to think toward it. That is a more durable thing than memorizing any feature the language has to offer.

How to Use This Book

Do not try to memorize everything.

Focus on one part at a time and let each idea build on the previous one. This book is designed as a staircase, not a bag of unrelated topics. Values come first. Then naming. Then relationships. Then decisions. Then repetition. Then reuse. Then structure. Then delayed work, outside interaction, and larger systems.

If something feels confusing, keep going. Very often, the next piece makes the earlier one clearer.

Looking Forward

You have now seen the whole map. Every concept in this book has a place on it, and every place on it exists because something earlier made it necessary.

The map starts with values — but that raises a question worth sitting with before we move on.

What exactly is a value? Not in the vague sense, but precisely: what does it mean for something to be a value in a programming language? Why does that question have to be answered before anything else can exist?

It turns out to be a more interesting question than it first appears.

VALUES

---

the basic units of JavaScript

What a Value Is

JavaScript works with values.

A value is a unit of meaning the language can recognize and use. A number is a value. A piece of text is a value. A yes-or-no result is a value.

Values come first because they are the first problem a language has to solve. Before a language needs variables, expressions, conditions, or functions, it needs some basic unit of meaning it can represent and work with. Variables do not exist without values to refer to. Expressions do not exist without values to combine. Conditions do not exist without values to test.

Values are the raw material. Everything else is built on top of them.

Once that is clear, the next question becomes obvious: are all values the same kind of thing?

Value Types

No. Not all values are the same kind of thing.

Some values represent text, like "hello". Some represent quantity, like 42. Some represent a yes-or-no state, like true or false. Some represent absence, like null or undefined.

All of these are values. They are not all the same kind of value, and that difference matters.

JavaScript groups values into types. A type is the category a value belongs to.

A value is what something is. A type is what kind of value it is.

For example:

42 is a value. number is its type. "hello" is a value. string is its type. false is a value. boolean is its type.

Type is not a label pasted on after the fact. It is part of what the value already is. "5" is not a number wearing quotation marks. It is text. true is not a string that happens to spell the word true. It is one of exactly two boolean values — true or false — that represent a yes-or-no state.

Type shapes what can happen with a value. A number can be used as quantity. A string can be combined as text. A boolean represents a state that is either on or off. These are not just different values — they are different kinds of values, and JavaScript handles them accordingly.

Similar-Looking Values Can Mean Different Things

Consider these three values:

```
"5" 5 true
```

They are all small and simple. They do not mean the same thing.

"5" is text. 5 is a number. true is a boolean.

That means JavaScript cannot treat them interchangeably.

5 + 2 produces 7 — quantity added to quantity. "5" + "2" produces "52" — text joined to text.

And true is neither text nor quantity. It represents a yes-or-no state, not something to add or concatenate.

Before JavaScript can work with a value, it has to know what kind of value it is dealing with. That is what type answers.

Mental Model

A value is a piece of data. Its type is the kind of data it is.

The value answers: what is it? The type answers: what kind of thing is it?

Common Pitfall

It is easy to assume that two values with similar-looking characters are the same kind of thing. They are not always.

"5" and 5 look related. They are different types. One is text. One is quantity. JavaScript treats them differently because they are different — not because of how they look, but because of what they are.

Syntax

```
42 "hello" true null undefined
```

These are all values. They belong to different types.

Chapter Summary

JavaScript works with values. A value is a unit of meaning the language can recognize and use — a number, a piece of text, a yes-or-no result, a representation of absence.

Values come first because everything else depends on them. Variables refer to values. Expressions combine them. Conditions test them. Without values, none of those ideas have anything to work with.

Not all values are the same kind of thing. JavaScript groups values into types, and type shapes what a value means and what can be done with it. A number is not the same as a string that looks like a number. A boolean is not a string that spells the word true. Type is not a label added after the fact — it is part of what the value already is.

Once Sentence: A value is the basic unit of meaning in JavaScript — the raw material everything else is built from.

Looking Forward

So far, we have looked at values from the side of meaning — what a value is, and what kind of value it is.

The next step is to look at how JavaScript treats different kinds of values differently under the hood.

Some values are basic and standalone. Others are structured with properties. JavaScript does not treat those the same way, because they are not the same kind of thing.

That leads directly to primitive values

PRIMITIVE VALUES

---

basic standalone values

Concept

JavaScript values come in two fundamentally different kinds.

Some values are basic and standalone. A number is a number. A string is a string. A boolean is either true or false. These values do not have named properties. They do not hold structure. JavaScript treats them as complete on their own.

These are called primitive values.

Other values are structured. They can contain properties, group related data, and hold callable logic. These are objects.

Both kinds are real values. The difference is not complexity — it is organization. A primitive is a single basic value. An object is a value with internal structure.

That distinction matters because JavaScript does not treat them the same way — and understanding why explains behavior that would otherwise feel strange.

The Core Difference

Think of a primitive as a single note. Think of an object as a chord.

A note stands on its own. A chord is multiple notes grouped together and treated as one musical unit. Both are real. Both are meaningful. One is a single sound. The other has internal structure.

```
42              // primitive — a single value
{ count: 42 }   // object — a value with structure
```

Arrays and functions are objects too. The object category is broader than curly-brace structures — it includes any value with internal structure.

Primitive Values

The core primitive types are:

- **number** → quantity
- **string** → text
- **boolean** → true or false
- **null** → intentional absence
- **undefined** → missing value

A primitive can contain many characters and still be primitive. A string like "hello" has five characters but JavaScript treats it as one basic value — not as a structured thing with named parts.

Number

A number represents quantity.

```
42
3.14
-10
```

Numbers are used for amounts, measurements, counts, and calculated results. They can be added, subtracted, compared, and calculated with.

The meaning of a number is quantity — not characters that look numeric. That is why `5` and `"5"` are not the same thing. One is quantity. The other is text.

String

A string represents text.

```
"hello"
"Ava"
"42"
```

Strings are used for words, names, sentences, and labels. The quotes tell JavaScript: treat this as text.

Even when the text inside looks like a number, it is still a string. `"5"` is text. `5` is quantity. Adding numbers and joining strings are not the same operation, because the values are not the same kind of thing.

Boolean

A boolean is one of exactly two values — `true` or `false` — used to represent a yes-or-no state.

```
true
false
```

Booleans are used when a program needs to represent whether something is the case: is the user signed in? Is the score high enough? Did the comparison match?

They become especially important when the program starts making decisions — which is the next major idea in this book.

Null

`null` represents intentional absence.

```
null
```

`null` means: there is no value here, and that absence is deliberate. It is not an error. It is not broken code. It is a real value with a specific meaning — this is empty on purpose.

Undefined

`undefined` represents a missing value.

undefined

`undefined` means a value has not been provided or has not been assigned yet.

The difference from `null` is subtle but worth holding onto:

- `null` → empty on purpose
- `undefined` → not provided yet

Both point toward absence. They are not the same kind of absence.

Why JavaScript Treats Them Differently

This is where the primitive-object distinction has real consequences.

Primitives behave independently.

When one name is assigned from another, and both refer to primitive values, the two names end up independent. Changing what one name refers to has no effect on the other.

```
let a = 5;
let b = a;
```

a = 10;

```
// a is 10, b is still 5
```

Objects do not work this way.

When one name is assigned from another, and both refer to an object, both names refer to the *same* object. A change made through one name is visible through the other.

```
let a = { count: 5 };
let b = a;
a.count = 10;
// b.count is also 10
```

This surprises many beginners because it is different from how primitives behave. It is not accidental — it is the direct result of how primitives and objects are organized differently.

Mental Model

A primitive is a single standalone value. An object is a structured value with internal parts.

The key question when reasoning about behavior: is this a primitive or an object?

That answer determines whether two names end up independent or whether they share the same underlying thing.

Common Pitfall

If one variable refers to an object and another is assigned from it, it is natural to assume the second gets its own copy. It does not. Both refer to the same object, and a change through one is visible through the other.

With primitives, two names end up independent. With objects, two names can refer to the same thing.

Challenge

Goal: Predict what each piece of code produces, and explain why.

What it builds on: Primitive types, the primitive-object distinction, and how names refer to values.

Part one

```
let a = 5;
let b = a;
a = 10;
console.log(b);
```

What does `console.log(b)` produce? Did `a`'s change affect `b`?

Part two

```
let a = { count: 5 };
let b = a;
a.count = 10;
console.log(b.count);
```

What does `console.log(b.count)` produce? Why is this different from part one?

Part three

```
let x = null;
let y;
console.log(x);
console.log(y);
```

Both `x` and `y` seem empty. Do they produce the same result? What is the difference?

Part four

```
console.log(typeof null);
console.log(typeof undefined);
console.log(typeof 42);
console.log(typeof "hello");
```

What does each line produce?

Answers

Part one

`console.log(b)` produces `5`. When `b` was assigned from `a`, both referred to the primitive value `5`. Primitives behave independently — reassigning `a` to `10` had no effect on `b`. The two names ended up independent.

Part two

`console.log(b.count)` produces `10`. Both `a` and `b` refer to the same object. When `a.count` was changed to `10`, that change is visible through `b` because both names point to the same underlying structure. This is the key difference from primitives.

Part three

`console.log(x)` produces `null`. `console.log(y)` produces `undefined`. They are both forms of absence — but different kinds. `null` was assigned deliberately: this is empty on purpose. `undefined` was never assigned: this name exists but has no value yet. JavaScript keeps them separate because the intent behind each is different.

Part four

`typeof null` produces `"object"` — a well-known quirk of JavaScript. `null` is a primitive, but `typeof` reports it as `"object"` for historical reasons. This is generally considered a bug in the language that was never fixed to preserve backward compatibility.

`typeof undefined` produces `"undefined"`.

`typeof 42` produces `"number"`.

`typeof "hello"` produces `"string"`.

Syntax

```
42            // primitive
{ count: 42 } // object
```

Chapter Summary

Primitive values are the simplest kind of values in JavaScript — basic, standalone, without internal structure. Objects are values with structure that can contain properties and group related data.

Primitives behave independently when assigned between names. Objects do not — two names can refer to the same object, and a change through one is visible through the other.

The six primitive types are: `number`, `string`, `boolean`, `null`, `undefined`, and `symbol` (covered later). Each represents a different kind of meaning. `null` and `undefined` both represent absence, but different kinds of absence.

One sentence: A primitive is a single basic value — without structure, without properties, complete on its own.

Looking Forward

Primitive values stand on their own — but a value on its own is not enough.

A program often needs to come back to a value later. It needs to refer back to it, compare it, use it in new expressions, or build on it.

The next question is not only what kinds of values exist. It is how we keep track of a value once we have one.

That leads directly to variables.

VARIABLES

---

keeping track of a value

The Problem

A value can appear in a program without having a name.

42

JavaScript recognizes that value. It knows `42` is a number. But without a name, the program has no stable way to refer back to it. It can appear, matter for a moment, and vanish from use.

A value on its own cannot be reused. It cannot be compared to something later. It cannot carry meaning across multiple steps of the code.

That is the problem variables solve.

What Is a Variable?

A variable is a name the program can use to refer to a value.

```
let x = 42;
```

There are several pieces here:

- `x` is the name
- `42` is the value
- the variable is the named thing in the program — the stable way to refer back to that value

The value and the variable are not the same thing. The value is the data. The variable is the name the program uses to reach it.

When we write `let x = 42`, the number `42` does not change. What changes is the program's structure around it. The code now includes a name that can be used to refer back to that value.

Naming Changes the Situation

A raw value is just a value.

42

Once a variable is introduced, the situation changes.

```
let score = 42;
```

Now the code has a name it can return to. The same value can appear in multiple places — not because it was copied and rewritten, but because the name travels with it.

Naming does not change the value itself. It changes the role the value can play in the program. Before naming, the value could only exist where it was written. After naming, the program can return to it, compare it, pass it somewhere else, and build on it.

That is why variables matter so much. They create persistence. They allow meaning to continue beyond the moment in which it first appeared.

`const` and `let`

JavaScript provides two modern ways to introduce a variable: `const` and `let`. They are not interchangeable.

`const` introduces a name that cannot be reassigned.

```
const pi = 3.14;
pi = 3;   // error — pi cannot be reassigned
```

Once a `const` name is associated with a value, that association is fixed. The name cannot be pointed at a different value.

`let` introduces a name that can be reassigned.

```
let score = 10;
score = 20;   // allowed — score now refers to 20
```

The name stays the same. What it refers to can change.

Which one to use?

Default to `const`. Use `let` only when you know the name will need to refer to a different value later — a counter that increments, a status that flips, a value that gets updated over time.

This is not just a style preference. Using `const` by default makes the program's intentions clearer. When a reader sees `const`, they know: this name will always refer to the same value. When they see `let`, they know: this name may change. That distinction carries real information.

A Note on `var`

There is an older way to declare variables in JavaScript: `var`.

```
var score = 10;
```

You will see it in older code. It works — programs ran on it for decades — but its scoping rules are different from `let` and `const` in ways that cause subtle bugs. Modern JavaScript uses `let` and `const` instead.

We will return to `var` in the Scope chapter, where its behavior can be explained fully. For now: if you see it, know what it is. In your own code, use `const` or `let`.

Declaration, Assignment, and Reassignment

These ideas are related but not identical.

Declaration introduces a name into the program:

```
let score;
```

Assignment gives the name something to refer to:

```
score = 10;
```

Declaration and initial assignment together:

```
let score = 10;
```

Reassignment points the name at a different value:

```
score = 20;
```

After reassignment, `score` refers to `20`. The name stayed the same. What it referred to changed.

That is the heart of why it is called a variable. The value can vary. The name remains available.

Note that `const` does not allow reassignment. Declaration and initial assignment must happen together, and the association is then fixed:

```
const name = "Ava";
// declaration and assignment together — required for const

name = "Avalynn";
// error — const cannot be reassigned
```

---

undefined

If a variable is declared without being assigned a value, it exists — but it refers to a special value called `undefined`.

```
let x;
console.log(x);   // undefined
```

`undefined` is not an error. It is JavaScript's way of saying: this name exists in the program, but no value has been given to it yet.

It is worth sitting with this for a moment, because it surprises beginners.

The variable `x` is real. It is in the program. JavaScript knows about it. But because no value was associated with it, JavaScript fills that gap with `undefined` — a value that means: not yet provided.

This is different from a variable that does not exist at all. If you try to use a name that was never declared, JavaScript will throw a reference error. If you use a name that was declared but not assigned, JavaScript will give you `undefined`.

```
console.log(x);
// ReferenceError - x was never declared
let x;
console.log(x);
// undefined - x exists but has no value yet
```

`undefined` will appear again throughout the language — in functions that return nothing, in object properties that don't exist, in parameters that weren't passed. Recognizing it as "not yet provided" rather than "broken" is one of the early shifts that makes JavaScript easier to reason about.

Mental Model

Think of a variable as a label attached to a string. The string connects the label to a value. The label does not change — but the string can be untied from one value and retied to another.

```
let score = 10;   // label "score" connected to 10
score = 20;       // label "score" reconnected to 20
```

`const` means the string cannot be retied. The label is permanently connected to its first value.

One important clarification: a variable is not a container with a value stuffed inside it. It is a name — a stable reference point. That distinction becomes important later when values get more complex. Names can be pointed elsewhere. Containers suggest the value lives inside the variable. It does not.

Common Pitfall

Reassignment does not merge or transform values.

```
let score = 10;
score = 20;
```

After this code, `score` refers to `20`. The value `10` did not become `20`. The name `score` was simply pointed at a different value. Those are two separate moments. The variable remained. What it referred to changed.

`const` does not mean the value itself cannot change — only the association.

```
const items = [1, 2, 3];
items.push(4);   // allowed — the array itself changed
items = [];      // error — the name cannot be reassigned
```

`const` prevents reassignment of the name. It does not freeze the value the name refers to. If the value is an object or array, its contents can still be modified. This distinction becomes more important when objects are covered later.

Challenge

Goal: Predict what each piece of code produces, and explain why.

What it builds on: Values, types, and how names refer to values.

Part one

```
let score = 10;
score = 20;
console.log(score);
```

What does `console.log(score)` produce? What happened to the value `10`?

Part two

```
const name = "Ava";
name = "Avalynn";
```

What happens here, and why?

Part three

```
let x;
console.log(x);
```

What does this produce? Is it an error?

Part four

```
const items = [1, 2, 3];
items.push(4);
console.log(items);
```

Does this produce an error? Why or why not?

Answers

Part one

`console.log(score)` produces `20`. The name `score` was reassigned from `10` to `20`. The value `10` was not destroyed — it simply no longer has a name pointing to it. The variable remained. What it referred to changed.

Part two

This produces an error. `name` was declared with `const`, which means the association is fixed. The name cannot be pointed at a different value.

Part three

This produces `undefined`. The variable `x` was declared but never assigned a value. JavaScript fills that gap with `undefined` — not an error, but a signal that the name exists and no value has been provided yet.

Part four

This does not produce an error. `items` was declared with `const`, which prevents the name from being reassigned to a different array. But the array itself — the value `items` refers to — can still be modified. `push` modifies the existing array. It does not reassign the name. The result is `[1, 2, 3, 4]`.

Why This Matters

Variables make values available for later use.

Without them, values can appear but cannot persist across multiple steps of the code. With them, the same value can be reused, compared, and built on over time.

`const` and `let` are not interchangeable. Choosing between them is one of the first places where a programmer's intentions become visible in the code itself.

Programming is not only about values existing. It is about values remaining available — and about making clear, through the choice of `const` or `let`, what is meant to stay fixed and what is meant to change.

Looking Forward

Now that values can be named and referred to, the next question is what the language can actually do with them.

Values can be added together, compared, associated with names, and combined into larger checks. Something has to sit between the values and make that happen.

Those are operators. That is the next step.