The Basics

This chapter takes a quick trip through a variety of different JavaScript topics. Some of them require knowledge about things we haven't covered yet. This "chicken or egg" problem means that this chapter has an unusual number of references to topics we'll discuss later. Don't let that throw you, though: we'll provide you with the essential information you need right now. The rest should become clear as you move through this book. In the meantime, pay attention to the aspects we identify as important and don't sweat the details.

**[Data Types](https://launchschool.com/books/javascript/read/basics" \l "datatypes)**

A core job of most programming languages is dealing with data. Different **data types** represent different kinds of data. Data types help programmers and their programs determine what they can and cannot do with a given piece of data.

JavaScript has five so-called **primitive data types**:

* String
* Number
* Undefined
* Null
* Boolean

Every type that is not a primitive type is an **object type**, which we'll discuss in more detail in a later chapter. We'll describe the different primitive types below.

ES6 (described later on) introduces two additional primitive types: Symbol and BigInt. We don't currently concern ourselves with these types in the Launch School curriculum, so you don't have to remember them.

You may notice that we sometimes use capitalized names when we're talking about types rather than individual values. For instance, we might refer to the Number type when we want to talk about numbers, in general, and their associated properties. However, if we're speaking of a numeric value, we'll generally refer to it as a number (uncapitalized).

The differences between these capitalized and uncapitalized names can be subtle and difficult to use properly. In general, though, the only time you really need to worry about it is in your code. In most of your code, you should use the capitalized name when you're referring to a built-in feature of JavaScript. The one exception to this rule is with the typeof operator -- it always returns a lowercase string value, so typeof 3 returns "number".

The remaining differences aren't important at this stage. You don't need to worry about the difference between a "number" and a "Number" as long as the rest of your language is clear. When to use the capitalized type names is generally not important, and many resources capitalize them inconsistently.

Data type values can be represented by **literals**. A literal is any notation that lets you represent a fixed value in source code. For instance, all of the following are literals in JavaScript:

Copy Code

'Hello, world!' // string literal

3.141592 // numeric literal

true // boolean literal

{ a: 1, b: 2 } // object literal

[ 1, 2, 3 ] // array literal

undefined // undefined literal

We will meet these literals and others soon.

Strings

A **string** is a list of characters in a specific sequence. In programming, we often have to work with text data like names, messages, and descriptions. JavaScript uses strings to represent such data. You write string **literals** with either single quotes ('hi there') or double quotes ("hi there") on either side of the text; note that the quotes are syntactic components, not part of the value.

Both syntactic forms create a string, but there are some subtle differences. Suppose you need a string that contains a single or double quote character as part of its value. In this case, you must find a way to include that quote character without having JavaScript treat it as syntax. For instance, you can include single quotes within your string by using double quotes around the entire string, or by escaping the single quote characters:

Copy Code

> "He said, 'Hi there!'" // with double quotes

= "He said, 'Hi there!'"

> 'He said, \'Hi there!\'' // with single quotes and escaping

= "He said, 'Hi there!'"

Note that Node displays the return values with double quotes here. It ordinarily uses single quotes, but it switches to double quotes when the string's value contains any single quotes.

A similar technique works when the string contains a double-quote character: put the entire string inside single quotes or escape the double-quote characters.

Copy Code

> 'He said, "Hi there!"' // with single quotes

= 'He said, "Hi there!"'

> "He said, \"Hi there!\"" // with double quotes and escaping

= 'He said, "Hi there!"'

This time Node uses single quotes; that's the default.

The backslash, or escape character (\), tells the computer that the next character isn't syntactic, but is part of the string. Escaping a quote character prevents JavaScript from seeing it as the end of the string. Pay attention to the type of slash you use: a forward slash (/) doesn't have any significance inside a string.

A recent JavaScript addition is template literals. They use backticks (`) and enable an operation called **string interpolation**. To see how it works, try the following code in node:

Copy Code

> `5 plus 5 equals ${5 + 5}`

= '5 plus 5 equals 10'

On keyboards with US layouts, the backtick typically appears on the upper-left portion of your keyboard, somewhere near the 1 key. It may be elsewhere or require modifier keys on non-US keyboards.

String interpolation is a handy way to merge JavaScript expressions with strings. The basic syntax is:

Copy Code

`Blah ${expression} blah.`

JavaScript replaces the ${expression} substring with the value of the expression inside the braces: it **interpolates** the expression's value. You'll learn to love this feature; it saves a lot of time and effort. However, remember that string interpolation is a feature of template literals (i.e., within backticks) and doesn't work in single- or double-quoted strings.

Numbers

The Number data type represents all kinds of numbers in JavaScript. Some programming languages have different data types for different number types, such as integers, floating-point numbers, and fixed-point (decimal) numbers. JavaScript has a single data type, Number, that represents all types of numbers. For the math whizzes out there, we mean real numbers.

Copy Code

1, 2, -3, 4.5, -6.77, 234891234 // Examples of numeric literals

Note that you can't use commas or periods for grouping: neither 123,456,789 nor 123.456.789 is valid. You must use 123456789 instead. Similarly, you can't use European-style period separators with a comma before the decimal part: use 543.21, not 543,21.

Recent versions of JavaScript let you use underscores for grouping in numbers, e.g., 12\_345\_678. However, as of 2022, support for this new feature hasn't yet reached a point where you can rely on it.

Booleans

**Boolean** values represent an "on" or "off" state. For example, if you want to represent the state of a light switch in your application, you can use boolean values. There are two boolean literal values: true and false:

Copy Code

> let toggleOn = true

= undefined

> let sessionActive = false

= undefined

Boolean values have a starring role when working with comparison operators. We'll talk about comparisons in the next section. For now, it's enough to know that comparisons check whether a value is equal to, less than, or greater than another value. They return a boolean result (true or false).

Copy Code

> 5 === 5

= true

> 100 < 99

= false

As you can see, comparison operators evaluate to either true or false. That === may look strange, but don't worry. We'll explain it soon.

Undefined

In programming, we need a way to express the absence of a value. In JavaScript, we do this with the value undefined. When a variable is not defined, its value is given by undefined. We can describe undefined as representing the absence of a value. We can also explicitly use the literal undefined.

Copy Code

setAgeFor('Pete', undefined);

The console.log function is an example of a function that returns undefined:

Copy Code

> console.log("Hello, World!")

**Hello, World!**

= undefined

console.log writes a value to the console. However, it doesn't have a reason to return anything specific. Instead, it returns undefined.

Don't get too concerned if you don't understand what a function or return value is. We'll get there in the Functions chapter. For now, it's enough to know that console.log is a function and its return value is undefined. (A little later, we'll meet the Number and String functions.)

undefined also arises when declaring variables without an explicit value. We'll talk about variables in the next chapter, but, for now, you can think of them as labels that represent values. Here's an example in node:

Copy Code

> let foo

= undefined

> foo

= undefined

> let bar = 3

= undefined

> bar

= 3

As you can see, we declare the foo variable without giving it a value. Using that variable returns undefined. On the other hand, we declare bar with an initial value of 3. Thus, using bar in an expression returns 3.

Null

null is similar to undefined: it represents the intentional absence of a value. Often, null represents emptiness or nothing. The chief difference between null and undefined is that you must use null explicitly if you want to use it; undefined can arise implicitly. These two values are so similar in their use and behavior that some people think having both in JavaScript is a mistake. We'll explore null a little later. For now, you can think of it as a value that represents emptiness or nothing.

Copy Code

> let foo = null

In the above code, null is specified as a literal value.

The typeof Operator

Every value you use in your JavaScript programs has a data type. To see what type a particular value has, you can use the typeof operator. typeof returns a string that contains the type of its operand's value. For example:

Copy Code

> typeof 1

= 'number'

> typeof 'foo'

= 'string'

> typeof true

= 'boolean'

> typeof undefined

= 'undefined'

> typeof null

= 'object'

> typeof [1, 2, 3]

= 'object'

We can make a couple of observations here.

* We can see that the value undefined has type 'undefined'. The name of the type and the value are distinct. The 'undefined' type has one possible value: undefined. To make this clearer, we sometimes write Undefined (with a capital U) when referring to the type name. However, the typeof operator doesn't make this distinction.
* For some bizarre reason, null's type is 'object'. No, that's not a typo: typeof null returns 'object'! This oddity is, in fact, a mistake in JavaScript itself. It goes back to the first version of the language, which took but 10 days to develop. Subsequent implementations preserved the error, and now we're stuck with it. Fixing it would break too many applications.

If you search the Internet, you'll eventually find some sites that claim that null is actually an object -- primarily because of this issue. Even MDN says it is an object on at least one page, but elsewhere claims it is a primitive value. In the end, the ECMAScript standards document describes null as a primitive value. Therefore, you should treat it as a primitive value.

* Another peculiarity is that the type of an array ([1, 2, 3] is an example) is also 'object'. This makes some sort of sense since, in JavaScript, arrays *are* objects. They just have some special characteristics. (We will meet arrays later and revisit this point.)

**[Operations](https://launchschool.com/books/javascript/read/basics" \l "operations)**

In the previous chapter, we learned that programming languages have data types and operations that you can perform on those data types. We've looked at JavaScript's most basic data types so far: the primitive types. Let's turn our attention now to some elementary operations that you can perform on the primitive types.

Adding, Subtracting, and Multiplying numbers

Basic arithmetic operations in JavaScript are simple. Make sure you try the examples yourself and take a few minutes to play with other numbers so you can get a feel for using them.

Use the + operator to perform addition:

Copy Code

> 38 + 4

= 42

Subtraction uses the '-' operator:

Copy Code

> 44 - 2

= 42

Multiplication uses the \* operator:

Copy Code

> 7 \* 6

= 42

Division requires a bit more depth, so we've set aside a separate section for it.

Division and Remainder

You can divide two numbers—integers or decimals—with the / operator.

Copy Code

> 16 / 4

= 4

> 16 / 2.5

= 6.4

When you divide two integers that don't result in a whole number, you get a decimal number in the result:

Copy Code

> 16 / 5

= 3.2

> 16 / 7

= 2.2857142857142856

JavaScript also has a **remainder operator**, %. It returns the remainder of a division operation. Let's try it:

Copy Code

> 16 % 4

= 0

That makes sense since 16 is evenly divisible by 4, so the remainder is 0. It's not an interesting example, in any case. Let's try another:

Copy Code

> 16 % 5

= 1

There we go! Since 16 divided by 5 equals 3 with a remainder of 1, this operation returns 1.

The remainder operator also works with decimal numbers, though it's a little hard to imagine a use case for it.

Note that JavaScript's % operator computes the remainder of dividing two numbers; it does not compute the modulo value, nor does it have any built-in methods that will compute the modulo value. In mathematics, there is a subtle, but important, difference between modulo and remainder operations. We won't get into the [mathematical definitions](https://math.stackexchange.com/questions/801962/difference-between-modulus-and-remainder), but the effect can be summarized as follows:

* Remainder operations return a positive integer when the first operand is positive, and a negative integer when the first operand is negative.
* Modulo operations return a positive integer when the second operand is positive, and a negative integer when the second operand is negative.

We can see this definition at work in this table:

| **a** | **b** | **a % b (remainder)** | **a modulo b** |
| --- | --- | --- | --- |
| 17 | 5 | 2 | 2 |
| 17 | -5 | 2 | -3 |
| -17 | 5 | -2 | 3 |
| -17 | -5 | -2 | -2 |

In case you're wondering why some of the values in the modulo column have an absolute value of 2 but others have an absolute value of 3, the main thing to observe is that 5 - 2 == 3: 5 is the absolute value of b (the modulus).

When working with positive integers, we don't need to worry whether we should use the remainder operator or (somehow) compute a modulo value; the result is positive and identical either way. However, if either number is negative, that's where things get confusing and may lead to problems. It really isn't easy to keep the above rules in mind, especially when you don't encounter them often. To complicate matters even more, JavaScript does not have a built-in modulo operator or method, and some languages use the % operator to compute the modulo value instead of the remainder.

In general, you want to avoid this problem -- if you need to determine the modulo or remainder of two integers, try to work with positive integers exclusively. If you can't, then make sure you know exactly what the operator does when working with negative integers.

NaN

Some operations, such as 0 / 0, return the value NaN, which stands for "Not a Number." This special value signals an illegal operation on numbers, such as 0 / 0, or some other operation that doesn't make sense, such as 3 + undefined.

All values in JavaScript have a type. What, then, is NaN's type? Let's see:

Copy Code

> typeof NaN

= 'number'

Wait. What? How can "Not a Number" be a number? It's a little hard to comprehend, but it helps to think of NaN as a numeric result that indicates an error occurred. NaN values arise in two main situations:

* Undefined mathematical operations, such as dividing 0 by 0 or trying to take the square root of a negative number, return NaN. Note that *undefined* here is a mathematical term: it doesn't refer to JavaScript's undefined value but to a mathematical operation that isn't defined.
* Trying to convert a non-number value, such as 'hello', to a number can also return NaN.

To determine whether a value is NaN, you can't use the usual comparison operators in a simple way. As it happens, NaN is the only value in JavaScript that is not equal to itself:

Copy Code

> let value = NaN;

> value === NaN // We'll talk about this in a few minutes

= false

> NaN === NaN

= false

Instead, you should use either Number.isNaN or Object.is:

Copy Code

> let value = NaN;

> Number.isNaN(value)

= true

> Object.is(value, NaN)

= true

Infinity and -Infinity

Other mathematical operations, such as 1 / 0, return Infinity, which indicates a number of infinite magnitude. How is that different from NaN? For most practical purposes, it doesn't matter whether you have NaN or Infinity, but they are very different concepts. Infinity is a number that is so large, it can't be written down. If you start counting 1, 2, 3, 4, and so on, saying a new number every second and keep going for the next 37 trillion years, you will not ever reach the value of Infinity. There is always a larger value. NaN, on the other hand, is the result of an attempted mathematical operation that is neither a valid number nor an infinite number.

One common operation that results in Infinity is to divide a positive number by 0. For instance, 1 divided by 0 (written 1 / 0) returns Infinity. There are many other operations that can return Infinity, but division by 0 is the one you're most likely to run into.

Infinity has some other interesting, and sometimes puzzling, properties:

Copy Code

> Infinity \* Infinity

= Infinity

> Infinity + Infinity

= Infinity

> Infinity - Infinity

= NaN

> Infinity / Infinity

= NaN

> 1234567890 / Infinity

= 0

Infinity has a negative counterpart, -Infinity, that represents an infinite value that is less than 0. It most commonly arises when dividing a negative number by 0:

Copy Code

> -1 / 0

= -Infinity

As with NaN, Infinity and -Infinity are considered by JavaScript to be numbers:

Copy Code

> typeof Infinity

= 'number'

> typeof -Infinity

= 'number'

You can safely use comparison operators like === (see the next section) to determine whether a value is Infinity or -Infinity:

Copy Code

> let value1 = Infinity

> value1 === Infinity

= true

> let value2 = -Infinity

> value2 === -Infinity

= true

However, be wary of using expressions in such comparisons:

Copy Code

> let value1 = Infinity

> value1 === Infinity \* 123456789

= true

Equality Comparison

You sometimes want to determine whether two values are identical. For that, you can use the === operator. It compares two operands for equality and returns true or false as appropriate.

Let's try some comparisons in Node. Don't forget to enter these examples!

Copy Code

> 42 === 42

= true

> 42 === 43

= false

> 'foo' === 'foo' // It works with strings too

= true

> 'FOO' === 'foo' // Case is different

= false

If you've seen programs in other programming languages, you might be surprised by the === operator. Most other programming languages use == (two = characters). JavaScript also has an == operator, but its behavior is a bit odd compared to other languages. We'll discuss those details a little later. For now, avoid the == operator and use === instead.

JavaScript has some more comparison operators that we'll discuss in the *Flow Control* chapter.

String Concatenation

String concatenation looks like arithmetic addition, but the result is entirely different. When you use the + operator to join two strings, you are concatenating their values. Back to node!

Copy Code

> 'foo' + 'bar'

= 'foobar'

That looks simple enough, but feel free to play around with this functionality to get a better feel for it. You might run into some surprises.

Here's one of the surprises. What will the following code return? Try answering before you try it.

Copy Code

> '1' + '2'

If you thought it would return 3, that makes sense. However, since '1' and '2' are both strings, JavaScript performs concatenation instead. That's why the result is '12'.

What happens if we try to concatenate a string with a number?

Copy Code

> '1' + 2

= '12'

JavaScript concatenates the two values as though they were both strings. First, it converts—**coerces**—the number 2 to a string; it then concatenates the result to the string '1'. We call this behavior **implicit type coercion**. When using +, if either operand is a string and the other is not, JavaScript coerces the non-string operand to a string; thus, the result is always another string. We'll discuss coercion in some more detail later.

What happens, though, when you perform some other arithmetic operation on a string and number?

Copy Code

> '5' - 3

= 2

**[Explicit Coercion](https://launchschool.com/books/javascript/read/basics" \l "explicitcoercion)**

Suppose you have two string values in your program, '1' and '2', that you want to add mathematically and get 3. You can't do that directly since + performs concatenation when one of its operands is a string. Somehow, we need to coerce the strings '1' and '2' to the numbers 1 and 2: we want to perform an **explicit type coercion**. The difference between explicit and implicit coercion is that explicit coercion lets you decide what you want to do, whereas implicit coercion lets the engine choose.

Strings to Numbers

The Number function coerces a string to a number:

Copy Code

> Number('1')

= 1

Number takes a string value as an argument and returns a number if the string contains a valid numeric value. You can then perform arithmetic operations on the result.

What happens if we use Number on a non-numeric string?

Copy Code

> Number('foo')

= NaN

Interesting! JavaScript doesn't issue an error message. Instead, it returns NaN. This kind of thing happens a lot in JavaScript. In situations where most other languages raise an error, JavaScript fails silently. The programmer must determine whether an error represents a problem.

You can also use the parseInt function to coerce strings to numbers. As the name suggests, parseInt parses an integer from a string. Let's see an example:

Copy Code

> parseInt('12')

= 12

A surprising feature of parseInt is how it handles a string whose value begins with a numeric digit or a + or - sign followed by a digit. In both cases, it returns a numeric value, even though the remaining characters may not be numeric. It stops converting and ignores everything else once it encounters an invalid character:

Copy Code

> parseInt('12xyz')

= 12

This behavior also explains why parseInt returns an integer when the string is a number with a fractional component:

Copy Code

> parseInt('3.1415')

= 3

If the number in the string is too big or too small to represent as a JavaScript number, parseInt returns Infinity or -Infinity. JavaScript can handle some immensely large and small numeric values. However, at some point, it runs up against the limitations of its internal number representation. For parseInt, that limit occurs when the string contains a little more than 300 digits. We won't bother you with an example. At that point, parseInt returns a number that represents infinity.

A similar function, parseFloat, coerces a string to a floating-point (decimal) number.

Copy Code

> parseFloat('12.5foo')

= 12.5

Numbers to Strings

You can also coerce numbers into strings. The String function provides this capability. Let's see an example:

Copy Code

> String(20)

= '20'

We've merely scratched the surface of JavaScript coercion. It's an involved topic that we discuss in much more detail in the Core Curriculum. For now, this section provides everything you need to know about coercing strings to numbers and vice versa.

**[Data Structures](https://launchschool.com/books/javascript/read/basics" \l "datastructures)**

The two most common data structures, or complex data types, that JavaScript programmers use are arrays and objects. We'll cover them in more depth later, but an initial taste of both should help you get ready for the complete discussion.

Arrays

JavaScript organizes information into ordered lists using **arrays**. They may contain strings, numbers, booleans, or any other data type. In JavaScript, array literals use square brackets [ ] surrounding a comma-delimited list of values, otherwise known as elements. Let's make an array in node:

Copy Code

> [1, 2, 3, 4, 5]

**[ 1, 2, 3, 4, 5 ]**

This array contains the integers between 1 and 5. You can access an array element by placing its index inside brackets ([]). Don't get confused: JavaScript uses brackets both to define array literals and to access elements of an array. Don't worry: you'll get used to that overloaded syntax, and pretty soon. Element indexes are non-negative integers starting with 0. If we want the first element in the array shown above, we can do this:

Copy Code

> [1, 2, 3, 4, 5][0]

= 1

As you can see, the first element is at index 0. See what happens if you replace [0] with another index value. For instance, we can access the third element with index 2:

Copy Code

> [1, 2, 3, 4, 5][2]

= 3

Arrays can also be written in a multi-line format, which is especially useful for larger arrays or arrays with long values:

Copy Code

**[**

**"Eric Idle",**

**"John Cleese",**

**"Terry Gilliam",**

**"Graham Chapman",**

**"Michael Palin",**

**"Terry Jones",**

**]**

The trailing comma on the last item in the list is optional, but a common practice. Try to be consistent in its use.

The most important facts to remember about arrays are:

* The order of the elements is significant.
* Use index numbers to retrieve array elements.
* Index numbers are non-negative integers starting from 0.

Objects

JavaScript **objects** have many use cases, but the one that interests us most now is as a dictionary-like data structure that matches keys with specific values. Other languages use different names for the same kind of structure: hash, dictionary, and associative array are the most common terms. Essentially, a JavaScript object is a collection of key-value pairs.

You can create objects using object literals, which have zero or more key-value pairs separated by commas all embedded within curly braces ({}). A key-value pair associates a key and a given value. Each pair consists of a key, in the form of a string, and a value of any type. Key-value pairs in object literals use the key followed by a colon (:) and then the value.

That sounds more complicated than it is. Let's put those words into action. Type along!

Our first example creates an object with one key-value pair:

Copy Code

> { dog: 'barks' }

= { dog: 'barks' }

This object shows a key named dog, and the value associated with that key is the string 'barks'.

As with arrays, objects can have multiple elements. All we have to do is separate the key-value pairs with commas. However, unlike arrays, these values aren't in any specific order. Let's add some more items to our object:

Copy Code

> { dog: 'barks', cat: 'meows', pig: 'oinks' }

= { dog: 'barks', cat: 'meows', pig: 'oinks' }

Suppose we want to find out what noise a cat makes. We can retrieve a value by its key:

Copy Code

> ({ dog: 'barks', cat: 'meows', pig: 'oinks' })['cat']

= 'meows'

Note how similar the ['cat'] notation is to indexing with arrays. Instead of index values, though, we use key names in string form.

Objects can also be written in a multi-line format, which is especially useful for larger objects or objects with long or complex values:

Copy Code

**{**

**title: "Monty Python's Flying Circus",**

**cast: [**

**"Eric Idle",**

**"John Cleese",**

**"Terry Gilliam",**

**"Graham Chapman",**

**"Michael Palin",**

**"Terry Jones",**

**],**

**firstSeason: 1969,**

**lastSeason: 1974,**

**}**

The trailing comma on the last property in the object is optional, but a common practice. Try to be consistent in its use.

Objects are the building blocks of programming. You'll become intimately familiar with them as you learn more and more. For now, the most important thing to remember about objects is that you use keys to set or retrieve values.

Let's move on; we'll revisit these concepts later.

**[Expressions and Return Values](https://launchschool.com/books/javascript/read/basics" \l "expressionsandreturnvalues)**

When you use the Node or browser console REPL, you type **expressions** at the > prompt. An expression is anything that JavaScript can evaluate to a value, even if that value is undefined or null. With few exceptions, almost everything you write in JavaScript is an expression. JavaScript expressions evaluate to a value that can be captured and used in subsequent code.

JavaScript expressions are like expressions in mathematics; they evaluate to values that you can use as part of another expression:

Copy Code

> 7 + (5 + 2)

= 14

> console.log(5 + 2)

**7**

= undefined

In the above examples, 5 + 2 is an expression that evaluates to the value 7. In both cases, we use that value to continue evaluating the larger expression that contains 5 + 2.

You may have already noticed that when you enter an expression in a REPL, the REPL displays a value that you didn't specifically request. That's the evaluated value of your expression: the **return value**. In the previous example, the return values are 14 and undefined.

Expressions don't have to involve operators: any value is an expression that evaluates to itself:

Copy Code

> "hi"

= 'hi'

Printing (logging) vs returning values

Inexperienced JavaScript programmers are often confused with the difference between returning a value and printing or displaying it. When we invoke the console.log method, we're telling JavaScript to write something to the console. In Node, that is your screen; in your browser, it's the Console in your Developer Tools application. The term **log** is a synonym for printing or displaying something on the console.

Expressions do something, but they also return or evaluate to a value. The returned value may not always be what you expect. For instance, let's take a look at console.log:

Copy Code

> console.log('Howdy')

**Howdy**

= undefined

Here, console.log displayed "Howdy" on the console, but then it showed the word undefined in a different color or dimmer brightness. That extra output is the return value of the expression, and undefined tells you that console.log returned nothing. It's important to understand that distinction.

For example:

Copy Code

> let a = console.log("Howdy")

> a

What do we expect to happen here? The value returned by console.log("Howdy") is undefined, so that's the value to which a gets assigned. Therefore, a on the second line evaluates to undefined, and node shows it as the return value.

We'll explore return values with some depth later when we talk about functions. However, console.log is an essential tool, and we'll use it often. It's important to realize that the output and return value are different concepts.

**[Statements](https://launchschool.com/books/javascript/read/basics" \l "statements)**

JavaScript also has **statements**. You can find a complete list of statements on [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements). Statements often include expressions as part of their syntax, but the statement itself is not an expression -- its value cannot be captured and reused later in your code.

An example of a JavaScript statement is the variable declaration:

Copy Code

> let foo = 3;

On the other hand, the value 3 to the right of the = is an expression. If you later decide to print the value of foo, then foo in that code is also an expression:

Copy Code

> console.log(foo);

The key difference between a statement and an expression is that you can't capture a value from a statement. For instance, we can capture and print the value of the expression 3 \* 5 in the following call to console.log:

Copy Code

**console.log(3 \* 5);**

However, we can not replace 3 \* 5 with a let declaration or a while loop since both are statements:

Copy Code

**console.log(let answer = 3 \* 5);** // SyntaxError: missing ) after argument list

**console.log(while (true) {});** // SyntaxError: Unexpected token 'while'

The error message you see will vary based on how and where you try to misuse a statement.

Statements in Practice

The [ECMAScript Specification](https://262.ecma-international.org/10.0/#sec-ecmascript-language-statements-and-declarations) has an entire section devoted to defining what kinds of things are statements in JavaScript. For the most part, this list seems to agree with MDN's [list of statements](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements).

In practice, the term statement is used a little more broadly. In fact, this [MDN definition](https://developer.mozilla.org/en-US/docs/Glossary/Statement) says that:

a **statement** is a line of code commanding a task. Every program consists of a sequence of statements.

By this definition, every chunk of code that can be treated as a single unit is a statement. This includes:

* variable, function, and class declarations
* loops and if statements
* return and break statements
* assignments: a = 3;
* standalone expressions: console.log("Hello");

The chief difference between this latter definition and the language specification is it includes assignments and function calls as statements, while the language specification does not. Most developers use the term "statement" in the broader sense: any syntactic unit of code that expresses an action for the computer to perform.

If you're worried about keeping all this straight, **don't fret**. We'll use the term *statement* loosely, and so can you, provided your meaning is clear, precise, and unambiguous. However, try to keep the distinction in mind - expressions can be part of a statement, but not all statements can be part of an expression. The statements listed on the MDN page can't be part of an expression.

**[Summary](https://launchschool.com/books/javascript/read/basics" \l "summary)**

This chapter covered the basic building blocks of JavaScript. You learned about its basic data types and how to combine those types with operators. You also learned about data structures and how to use them to hold and access data. We'll go much deeper into these topics in the coming chapters, but, for now, let's learn with our fingers and do some exercises to deepen our understanding of the basics.

Some exercises, both here on this page and elsewhere in the curriculum, may use unfamiliar terms that aren't directly related to programming. For instance, we might refer to the squares of numbers, palindromes, and anagrams, all of which aren't directly pertinent to programming.

If you encounter an unfamiliar term, don't hestitate to search for their definitions elsewhere - in most cases, it's relatively easy to find a basic definition for the term. For instance, we may use the term "factorial" in an exercise - a google search for "factorial" should quickly lead you to a basic definition, such as [the one on this page](https://www.mathsisfun.com/numbers/factorial.html).

This is also good practice for the future. You may not always be writing code for a discipline that you're completely familiar with. For instance, if you're writing flight planning software, you may need to learn how to determine distances given the latitude and longitude of two points. (That is more complex than you might expect.) Fortunately, it's relatively easy to learn how to do so, even without a significant background in mathematics.