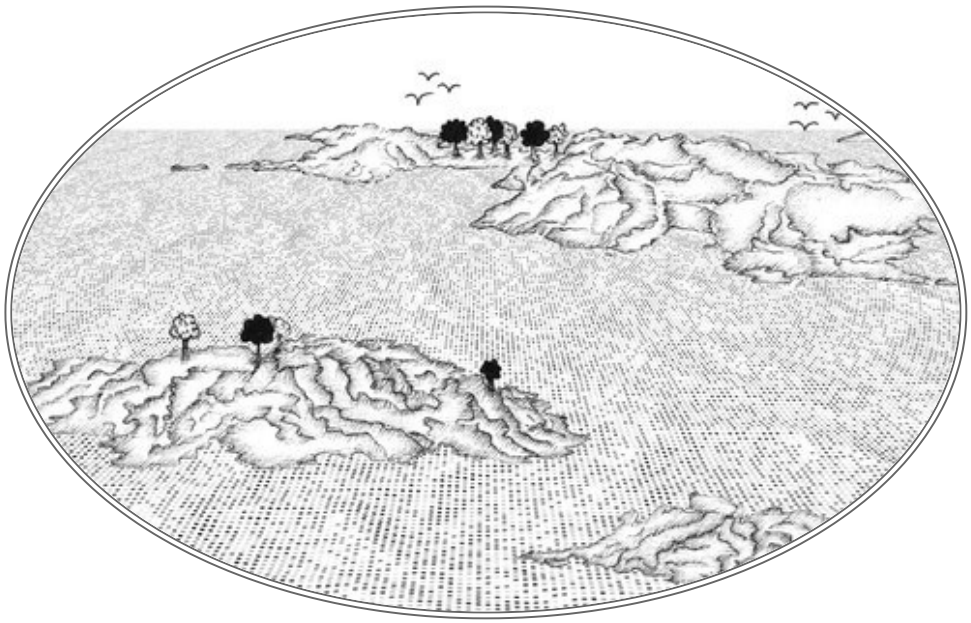
Values, Types, and Operators

> "Below the surface of the machine, the program moves. Without effort, it expands and ¶ contracts. In great harmony, electrons scatter and regroup. The forms on the monitor are but ripples on the water. The essence stays invisibly below."
>
> —Master Yuan-Ma, *The Book of Programming*



In the computer's world, there is only data. You can read data, modify data, create new data—but that which isn't data cannot be mentioned. All this data is stored as long sequences of bits and is thus fundamentally alike.

Bits are any kind of two-valued things, usually described as zeros and ones. Inside the computer, they take forms such as a high or low electrical charge, a strong or weak signal, or a shiny or dull spot on the surface of a CD. Any piece of discrete information can be reduced to a sequence of zeros and ones and thus represented in bits.

For example, we can express the number 13 in bits. This works the same way as a decimal number, but instead of 10 different digits, we have only 2, and the weight of each increases by a factor of 2 from right to left. Here are the bits that make up the number 13, with the weights of the digits shown below them:

```
  0   0   0   0   1   1   0   1
128  64  32  16   8   4   2   1
```

That's the binary number 00001101. Its nonzero digits stand for 8, 4, and 1, and add up to 13.

Values

Imagine a sea of bits—an ocean of them. A typical modern computer has more than 100 billion bits in its volatile data storage (working memory). Nonvolatile storage (the hard disk or equivalent) tends to have yet a few orders of magnitude more.

To be able to work with such quantities of bits without getting lost, we separate them into chunks that represent pieces of information. In a JavaScript environment, those chunks are called *values*. Though all values are made of bits, they play different roles. Every value has a type that determines its role. Some values are numbers, some values are pieces of text, some values are functions, and so on.

To create a value, you must merely invoke its name. This is convenient. You don't have to gather building material for your values or pay for them. You just call for one, and *whoosh*, you have it. Of course, values are not really created from thin air. Each one has to be stored somewhere, and if you want to use a gigantic number of them at the same time, you might run out of computer memory. Fortunately, this is a problem only if you need them all simultaneously. As soon as you no longer use a value, it will dissipate, leaving behind its bits to be recycled as building material for the next generation of values.

The remainder of this chapter introduces the atomic elements of JavaScript programs, that is, the simple value types and the operators that can act on such values.

Numbers

Values of the *number* type are, unsurprisingly, numeric values. In a JavaScript program, they are written as follows:

```
13
```

Using that in a program will cause the bit pattern for the number 13 to come into existence inside the computer's memory.

JavaScript uses a fixed number of bits, 64 of them, to store a single number value. There are only so many patterns you can make with 64 bits, which limits the number of different numbers that can be represented. With *N* decimal digits, you can represent 10^N numbers. Similarly, given 64 binary digits, you can represent 2^{64} different numbers, which is about 18 quintillion (an 18 with 18 zeros after it). That's a lot.

Computer memory used to be much smaller, and people tended to use groups of 8 or 16 bits to represent their numbers. It was easy to accidentally *overflow* such small numbers—to end up with a number that did not fit into the given number of bits. Today, even computers that fit in your pocket have plenty of memory, so you are free to use 64-bit chunks, and you need to worry about overflow only when dealing with truly astronomical numbers.

Not all whole numbers less than 18 quintillion fit in a JavaScript number, though. Those bits also store negative numbers, so one bit indicates the sign of the number. A bigger issue is representing nonwhole numbers. To do this, some of the bits are used to store the position of the decimal point. The actual maximum whole number that can be stored is more in the range of 9 quadrillion (15 zeros)—which is still pleasantly huge.

Fractional numbers are written using a dot:

```
9.81
```

For very big or very small numbers, you may also use scientific notation by adding an *e* (for *exponent*), followed by the exponent of the number.

```
2.998e8
```

That's $2.998 \times 10^8 = 299{,}800{,}000$.

Calculations with whole numbers (also called *integers*) that are smaller than the aforementioned 9 quadrillion are guaranteed to always be precise.

Unfortunately, calculations with fractional numbers are generally not. Just as π (pi) cannot be precisely expressed by a finite number of decimal digits, many numbers lose some precision when only 64 bits are available to store them. This is a shame, but it causes practical problems only in specific situations. The important thing is to be aware of it and treat fractional digital numbers as approximations, not as precise values.

Arithmetic

The main thing to do with numbers is arithmetic. Arithmetic operations such as addition or multiplication take two number values and produce a new number from them. Here is what they look like in JavaScript:

```
100 + 4 * 11
```

The `+` and `*` symbols are called *operators*. The first stands for addition and the second stands for multiplication. Putting an operator between two values will apply it to those values and produce a new value.

Does this example mean "Add 4 and 100, and multiply the result by 11", or is the multiplication done before the adding? As you might have guessed, the multiplication happens first. As in mathematics, you can change this by wrapping the addition in parentheses.

```
(100 + 4) * 11
```

For subtraction, there is the `-` operator. Division can be done with the `/` operator.

When operators appear together without parentheses, the order in which they are applied is determined by the *precedence* of the operators. The example shows that multiplication comes before addition. The `/` operator has the same precedence as `*`. Likewise, `+` and `-` have the same precedence. When multiple operators with the same precedence appear next to each other, as in `1 - 2 + 1`, they are applied left to right: `(1 - 2) + 1`.

Don't worry too much about these precedence rules. When in doubt, just add parentheses.

There is one more arithmetic operator, which you might not immediately recognize. The `%` symbol is used to represent the *remainder* operation. `X % Y` is the remainder of dividing `X` by `Y`. For example, `314 % 100` produces `14`, and `144 % 12` gives `0`. The remainder operator's precedence is the same as that of multiplication and division. You'll also often see this operator referred to as *modulo*.

Special numbers

There are three special values in JavaScript that are considered numbers but don't behave like normal numbers. The first two are `Infinity` and `-Infinity`, which represent the positive and negative infinities. `Infinity - 1` is still `Infinity`, and so on. Don't put too much trust in infinity-based computation, though. It isn't mathematically sound, and it will quickly lead to the next special number: `NaN`.

`NaN` stands for "not a number", even though it *is* a value of the number type. You'll get this result when you, for example, try to calculate `0 / 0` (zero divided by zero), `Infinity - Infinity`, or any number of other numeric operations that don't yield a meaningful result.

Strings

The next basic data type is the *string*. Strings are used to represent text. They are written by enclosing their content in quotes.

```
`Down on the sea`
"Lie on the ocean"
'Float on the ocean'
```

You can use single quotes, double quotes, or backticks to mark strings, as long as the quotes at the start and the end of the string match.

You can put almost anything between quotes to have JavaScript make a string value out of it. But a few characters are more difficult. You can imagine how putting quotes between quotes might be hard, since they will look like the end of the string. *Newlines* (the characters you get when you press ENTER) can be included only when the string is quoted with backticks (`` ` ``).

To make it possible to include such characters in a string, the following notation is used: a backslash (`\`) inside quoted text indicates that the character after it has a special meaning. This is called *escaping* the character. A quote that is preceded by a backslash will not end the string but be part of it. When an `n` character occurs after a backslash, it is interpreted as a newline. Similarly, a `t` after a backslash means a tab character. Take the following string:

```
"This is the first line\nAnd this is the second"
```

This is the actual text in that string:

```
This is the first line
And this is the second
```

There are, of course, situations where you want a backslash in a string to be just a backslash, not a special code. If two backslashes follow each other, they will collapse together, and only one will be left in the resulting string value. This is how the string "*A newline character is written like `"\n"`.*" can be expressed:

```
"A newline character is written like \"\\n\"."
```

Strings, too, have to be modeled as a series of bits to be able to exist inside the computer. The way JavaScript does this is based on the *Unicode* standard. This standard assigns a number to virtually every character you would ever need, including characters from Greek, Arabic, Japanese, Armenian, and so on. If we have a number for every character, a string can be described by a sequence of numbers. And that's what JavaScript does.

There's a complication though: JavaScript's representation uses 16 bits per string element, which can describe up to 2^{16} different characters. However, Unicode defines more characters than that—about twice as many, at this point. So some characters, such as many emoji, take up two "character positions" in JavaScript strings. We'll come back to this in Chapter 5.

Strings cannot be divided, multiplied, or subtracted. The `+` operator *can* be used on them, not to add, but to *concatenate*—to glue two strings together.

The following line will produce the string `"concatenate"`:

```
"con" + "cat" + "e" + "nate"
```

String values have a number of associated functions (*methods*) that can be used to perform other operations on them. I'll say more about these in Chapter 4.

Strings written with single or double quotes behave very much the same—the only difference lies in which type of quote you need to escape inside of them. Backtick-quoted strings, usually called *template literals*, can do a few more tricks. Apart from being able to span lines, they can also embed other values.

```
`half of 100 is ${100 / 2}`
```

When you write something inside `${}` in a template literal, its result will be computed, converted to a string, and included at that position. This example produces the string `"half of 100 is 50"`.

Unary operators

Not all operators are symbols. Some are written as words. One example is the `typeof` operator, which produces a string value naming the type of the value you give it.

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

We will use `console.log` in example code to indicate that we want to see the result of evaluating something. (More about that in the next chapter.)

The other operators shown so far in this chapter all operated on two values, but `typeof` takes only one. Operators that use two values are called *binary* operators, while those that take one are called *unary* operators. The minus operator (`-`) can be used both as a binary operator and as a unary operator.

```
console.log(- (10 - 2))
// → -8
```

Boolean values

It is often useful to have a value that distinguishes between only two possibilities, like “yes” and “no” or “on” and “off”. For this purpose, JavaScript has a *Boolean* type, which has just two values, true and false, written as those words.

Comparison

Here is one way to produce Boolean values:

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

The > and < signs are the traditional symbols for “is greater than” and “is less than”, respectively. They are binary operators. Applying them results in a Boolean value that indicates whether they hold true in this case.

Strings can be compared in the same way.

```
console.log("Aardvark" < "Zoroaster")
// → true
```

The way strings are ordered is roughly alphabetic but not really what you’d expect to see in a dictionary: uppercase letters are always “less” than lowercase ones, so `"Z" < "a"`, and nonalphabetic characters (!, -, and so on) are also included in the ordering. When comparing strings, JavaScript goes over the characters from left to right, comparing the Unicode codes one by one.

Other similar operators are `>=` (greater than or equal to), `<=` (less than or equal to), `==` (equal to), and `!=` (not equal to).

```
console.log("Garnet" != "Ruby")
// → true
```

```
console.log("Pearl" == "Amethyst")
// → false
```

There is only one value in JavaScript that is not equal to itself, and that is `NaN` ("not a number").

```
console.log(NaN == NaN)
// → false
```

`NaN` is supposed to denote the result of a nonsensical computation, and as such, it isn't equal to the result of any *other* nonsensical computations.

Logical operators

There are also some operations that can be applied to Boolean values themselves. JavaScript supports three logical operators: *and*, *or*, and *not*. These can be used to "reason" about Booleans.

The `&&` operator represents logical *and*. It is a binary operator, and its result is true only if both the values given to it are true.

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

The `||` operator denotes logical *or*. It produces true if either of the values given to it is true.

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

Not is written as an exclamation mark (`!`). It is a unary operator that flips the value given to it— `!true` produces `false` and `!false` gives `true`.

When mixing these Boolean operators with arithmetic and other operators, it is not always obvious when parentheses are needed. In practice, you can usually get by with knowing that of the operators we have seen so far, `||` has

the lowest precedence, then comes `&&`, then the comparison operators (`>`, `==`, and so on), and then the rest. This order has been chosen such that, in typical expressions like the following one, as few parentheses as possible are necessary:

```
1 + 1 == 2 && 10 * 10 > 50
```

The last logical operator we will look at is not unary, not binary, but *ternary*, operating on three values. It is written with a question mark and a colon, like this:

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

This one is called the *conditional* operator (or sometimes just *the ternary operator* since it is the only such operator in the language). The operator uses the value to the left of the question mark to decide which of the two other values to "pick". If you write `a ? b : c`, the result will be `b` when `a` is true and `c` otherwise.

Empty values

There are two special values, written `null` and `undefined`, that are used to denote the absence of a *meaningful* value. They are themselves values, but they carry no information.

Many operations in the language that don't produce a meaningful value yield `undefined` simply because they have to yield *some* value.

The difference in meaning between `undefined` and `null` is an accident of JavaScript's design, and it doesn't matter most of the time. In cases where you actually have to concern yourself with these values, I recommend treating them as mostly interchangeable.

Automatic type conversion

In the Introduction, I mentioned that JavaScript goes out of its way to accept almost any program you give it, even programs that do odd things. This is nicely demonstrated by the following expressions:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

When an operator is applied to the "wrong" type of value, JavaScript will quietly convert that value to the type it needs, using a set of rules that often aren't what you want or expect. This is called *type coercion*. The `null` in the first expression becomes `0` and the `"5"` in the second expression becomes `5` (from string to number). Yet in the third expression, `+` tries string concatenation before numeric addition, so the `1` is converted to `"1"` (from number to string).

When something that doesn't map to a number in an obvious way (such as `"five"` or `undefined`) is converted to a number, you get the value `NaN`. Further arithmetic operations on `NaN` keep producing `NaN`, so if you find yourself getting one of those in an unexpected place, look for accidental type conversions.

When comparing values of the same type using the `==` operator, the outcome is easy to predict: you should get true when both values are the same, except in the case of `NaN`. But when the types differ, JavaScript uses a complicated and confusing set of rules to determine what to do. In most cases, it just tries to convert one of the values to the other value's type. However, when `null` or `undefined` occurs on either side of the operator, it produces true only if both sides are one of `null` or `undefined`.

```
console.log(null == undefined);
// → true
```

```
console.log(null == 0);
// → false
```

That behavior is often useful. When you want to test whether a value has a real value instead of `null` or `undefined`, you can compare it to `null` with the `==` or `!=` operator.

What if you want to test whether something refers to the precise value `false`? Expressions like `0 == false` and `"" == false` are also true because of automatic type conversion. When you do *not* want any type conversions to happen, there are two additional operators: `===` and `!==`. The first tests whether a value is *precisely* equal to the other, and the second tests whether it is not precisely equal. Thus `"" === false` is false, as expected.

I recommend using the three-character comparison operators defensively to prevent unexpected type conversions from tripping you up. But when you're certain the types on both sides will be the same, there is no problem with using the shorter operators.

Short-circuiting of logical operators

The logical operators `&&` and `||` handle values of different types in a peculiar way. They will convert the value on their left side to Boolean type in order to decide what to do, but depending on the operator and the result of that conversion, they will return either the *original* left-hand value or the right-hand value.

The `||` operator, for example, will return the value to its left when that value can be converted to true and will return the value on its right otherwise. This has the expected effect when the values are Boolean and does something analogous for values of other types.

```
console.log(null || "user")
// → user
console.log("Agnes" || "user")
// → Agnes
```

We can use this functionality as a way to fall back on a default value. If you have a value that might be empty, you can put `||` after it with a replacement

value. If the initial value can be converted to false, you'll get the replacement instead. The rules for converting strings and numbers to Boolean values state that `0`, `NaN`, and the empty string (`""`) count as false, while all the other values count as true. That means `0 || -1` produces `-1`, and `"" || "!?"` yields `"!?"`.

The `??` operator resembles `||`, but returns the value on the right only if the one on the left is null or undefined, not if it is some other value that can be converted to `false`. Often, this is preferable to the behavior of `||`.

```
console.log(0 || 100);
// → 100
console.log(0 ?? 100);
// → 0
console.log(null ?? 100);
// → 100
```

The `&&` operator works similarly but the other way around. When the value to its left is something that converts to false, it returns that value, and otherwise it returns the value on its right.

Another important property of these two operators is that the part to their right is evaluated only when necessary. In the case of `true || X`, no matter what `X` is—even if it's a piece of program that does something *terrible*—the result will be true, and `X` is never evaluated. The same goes for `false && X`, which is false and will ignore `X`. This is called *short-circuit evaluation*.

The conditional operator works in a similar way. Of the second and third values, only the one that is selected is evaluated.

Summary

We looked at four types of JavaScript values in this chapter: numbers, strings, Booleans, and undefined values. Such values are created by typing in their name (`true`, `null`) or value (`13`, `"abc"`).

You can combine and transform values with operators. We saw binary operators for arithmetic (`+`, `-`, `*`, `/`, and `%`), string concatenation (`+`), comparison (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`), and logic (`&&`, `||`, `??`), as well

as several unary operators (`-` to negate a number, `!` to negate logically, and `typeof` to find a value's type) and a ternary operator (`?:`) to pick one of two values based on a third value.

This gives you enough information to use JavaScript as a pocket calculator but not much more. The next chapter will start tying these expressions together into basic programs.