CHAPTER 1

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*

Inside the computer's world, there is only data. You can read data, modify data, create new data—but anything that isn't data simply does not exist. 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 zeroes 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, think about how you might show the number 13 in bits. It works the same way you write decimal numbers, but instead of 10 different digits, you 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 after them:

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

So, that's 00001101, or 8 + 4 + 1, which equals 13.

VALUES

Imagine a sea of bits. An ocean of them. A typical modern computer has more than 30 billion bits in its volatile data storage. 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, you can 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. There are six basic types of values in JavaScript: numbers, strings, Booleans, objects, functions, and undefined values.

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 *woosh*, you have it. They are not created from thin air, of course. Every value has to be stored somewhere, and if you want to use a gigantic amount of them at the same time, you might run out of bits. 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.

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
```

Use that in a program, and it 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, namely, 64 of them, to store a single number value. There are only so many patterns you can make with 64 bits, which means that the amount of different numbers that can be represented is limited. For N decimal digits, the amount of numbers that can be represented is 10^N. Similarly, given 64 binary digits, you can represent 2^{64} different numbers, which is about 18 quintillion (an 18 with 18 zeroes after it). This is a lot.

Computer memory used to be a lot 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 amount of bits. Today, even personal computers have plenty of memory, so you are free to use 64-bit chunks, which means you need to worry about overflow only when dealing with truly astronomical numbers.

Not all whole numbers below 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 that nonwhole numbers must also be represented. 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 zeroes), which is still pleasantly huge.

Fractional numbers are written by using a dot.

```
9.81
```

For very big or very small numbers, you can also use scientific notation by adding an "e", followed by the exponent of the number:

```
2.998e8
```

That is $2.998 \times 10^8 = 299800000$.

Calculations with whole numbers (also called *integers*) 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 the 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. But, as in mathematics, you can change this by wrapping the addition in parentheses.

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

For subtraction, there is the `-` operator, and 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. `/` has the same precedence as `*`. Likewise for `+` and `-`. When multiple operators with the same

precedence appear next to each other (as in `1 - 2 + 1`), they are applied left to right.

These rules of precedence are not something you should worry about. When in doubt, just add parentheses.

There is one more arithmetic operator, which you might not 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`. Remainder's precedence is the same as that of multiplication and division. You'll often see this operator referred to as *modulo*, though technically *remainder* is more accurate.

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. It isn't mathematically solid, and it will quickly lead to our 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 precise, 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.

```
"Patch my boat with chewing gum"
'Monkeys wave goodbye'
```

Both single and double quotes can be used to mark strings as long as the quotes at the start and the end of the string match.

Almost anything can be put between quotes, and JavaScript will 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. *Newlines* (the characters you get when you press Enter) also can't be put between quotes. The string has to stay on a single line.

To be able to have such characters in a string, the following notation is used: whenever a backslash ("\") is found inside quoted text, it indicates that the character after it has a special meaning. 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"
```

The actual text contained is this:

```
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 written:

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

Strings cannot be divided, multiplied, or subtracted, but the `+` operator *can* be used on them. It does not add, but it *concatenates*—it glues two strings together. The following line will produce the string `"concatenate"`:

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

There are more ways of manipulating strings, which we will discuss when we get to methods in Chapter 4.

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. When you run such code, the value produced should be shown on the screen, though how it appears will depend on the JavaScript environment you use to run it.

The other operators we saw 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

Often, you will need a value that simply distinguishes between two possibilities, like “yes” and “no”, or “on” and “off”. For this, JavaScript has a *boolean* type, which has just two values: true and false (which are written simply as those words).

Comparisons

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 more or less alphabetic: uppercase letters are always “less” than lowercase ones, so `"Z" < "a"` is true, and nonalphabetic characters (!, -, and so on) are also included in the ordering. The actual comparison 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, Tamil, and so on. Having such numbers is useful for storing strings inside a computer because it makes it possible to represent them as a sequence of numbers. When comparing strings, JavaScript goes over them from left to right, comparing the numeric codes of the characters 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("Itchy" != "Scratchy")
// → true
```

There is only one value in JavaScript that is not equal to itself, and that is `NaN`

(which stands for "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 <= 100
```

The last logical operator I will discuss 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 *ternary* operator since it is the only such operator in the language). The value on the left of the question mark "picks" which of the other two values will come out. When it is true, the middle value is chosen, and when it is false, the value on the right comes out.

Undefined 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 (you'll see some in the next chapter) will 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 the cases where you actually have to concern yourself with these values, I recommend treating them as interchangeable (more on that in a moment).

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 these 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, it will quietly convert that value to the type it wants, using a set of rules that often aren't what you want or expect. This is called *type coercion*. So 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, the value `NaN` is produced. 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 outcome is easy to predict: you should get true when both values are the same. But when the types differ, JavaScript uses a complicated and confusing set of rules to determine what to do. I will not explain them precisely, but 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 last piece of behavior is often useful. When you want to test whether a value has a real value instead of `null` or `undefined`, you can simply compare it to `null` with the `==` (or `!=`) operator.

But what if you want to test whether something refers to the precise value `false`? 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`. Because of this, expressions like `0 == false` and `"" == false` are also true. For cases like this, where you do *not* want any automatic type conversions to happen, there are two extra operators: `===` and `!==`. The first tests whether a value is precisely equal to the other, and the second tests whether it is not precisely equal. So `"" === 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 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 can be converted to true and will return the value on its right otherwise. This conversion works as you'd expect for boolean values and should do something analogous for values of other types.

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

This functionality allows the `||` operator to be used as a way to fall back on a default value. If you give it an expression that might produce an empty value on the left, the value on the right will be used as a replacement in that case.

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 expression to their right is evaluated only when necessary. In the case of `true || X`, no matter what `X` is—even if it's an expression 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. The first expression is always evaluated, but the second or third value, the one that is not picked, is also not 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).

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