Chapter 3. Types, Values, and Variables

Computer programs work by manipulating values, such as the number 3.14 or the text "Hello World." The kinds of values that can be represented and manipulated in a programming language are known as types, and one of the most fundamental characteristics of a programming language is the set of types it supports. When a program needs to retain a value for future use, it assigns the value to (or "stores" the value in) a variable. Variables have names, and they allow use of those names in our programs to refer to values. The way that variables work is another fundamental characteristic of any programming language. This chapter explains types, values, and variables in JavaScript. It begins with an overview and some definitions.

3.1 Overview and Definitions

JavaScript types can be divided into two categories: *primitive types* and *object types*. JavaScript's primitive types include numbers, strings of text (known as strings), and Boolean truth values (known as booleans). A significant portion of this chapter is dedicated to a detailed explanation of the numeric (§3.2) and string (§3.3) types in JavaScript. Booleans are covered in §3.4.

The special JavaScript values `null` and `undefined` are primitive values, but they are not numbers, strings, or booleans. Each value is typically considered to be the sole member of its own special type. §3.5 has more about `null` and `undefined`. ES6 adds a new special-purpose type, known as Symbol, that enables the definition of language extensions without harming backward compatibility. Symbols are covered briefly in §3.6.

Any JavaScript value that is not a number, a string, a boolean, a symbol, `null`, or `undefined` is an object. An object (that is, a member of the type *object*) is a collection of *properties* where each property has a name and a value (either a primitive value or another object). One very special object, the *global object*, is covered in §3.7, but more general and more detailed coverage of objects is in Chapter 6.

An ordinary JavaScript object is an unordered collection of named values. The language also defines a special kind of object, known as an array, that represents an ordered collection of numbered values. The JavaScript language includes special syntax for working with arrays, and arrays have some special behavior that distinguishes them from ordinary objects. Arrays are the subject of Chapter 7.

In addition to basic objects and arrays, JavaScript defines a number of other useful object types. A Set object represents a set of values. A Map object represents a mapping from keys to values. Various "typed array" types facilitate operations on arrays of bytes and other binary data. The RegExp type represents textual patterns and enables sophisticated matching, searching, and replacing operations on strings. The Date type represents dates and times and supports rudimentary date arithmetic. Error and its subtypes represent errors that can arise when executing JavaScript code. All of these types are covered in Chapter 11.

JavaScript differs from more static languages in that functions and classes are not just part of the language syntax: they are themselves values that can be manipulated by JavaScript programs. Like any JavaScript value that is not a primitive value, functions and classes are a specialized kind of object. They are covered in detail in Chapters 8 and 9.

The JavaScript interpreter performs automatic garbage collection for memory management. This means that a JavaScript programmer generally does not need to worry about destruction or deallocation of objects or other values. When a value is no longer reachable—when a program no longer has any way to refer to it—the interpreter knows it can never be used again and automatically reclaims the memory it was occupying. (JavaScript programmers do sometimes need to take care to ensure that values do not inadvertently remain reachable—and therefore nonreclaimable—longer than necessary.)

JavaScript supports an object-oriented programming style. Loosely, this means that rather than having globally defined functions to operate on values of various types, the types themselves define methods for working with values. To sort the elements of an array `a`, for example, we don't pass `a` to a `sort()` function. Instead, we invoke the `sort()` method of `a`:

```
a.sort();       // The object-oriented version of sort(a).
```

Method definition is covered in Chapter 9. Technically, it is only JavaScript objects that have methods. But numbers, strings, boolean, and symbol values behave as if they have methods. In JavaScript, `null` and `undefined` are the only values that methods cannot be invoked on.

JavaScript's object types are *mutable* and its primitive types are *immutable*. A value of a mutable type can change: a JavaScript program can change the values of object properties and array elements. Numbers, booleans, symbols, `null`, and `undefined` are immutable—it doesn't even make sense to talk about changing the value of a number, for example. Strings can be thought of as arrays of characters, and you might expect them to be mutable. In JavaScript, however, strings are immutable: you can access the text at any index of a string, but JavaScript provides no way to alter the text of an existing string. The differences between mutable and immutable values are explored further in §3.8.

JavaScript liberally converts values from one type to another. If a program expects a string, for example, and you give it a number, it will automatically convert the number to a string for you. And if you use a non-boolean value where a boolean is expected, JavaScript will convert accordingly. The rules for value conversion are explained in §3.9. JavaScript's liberal value conversion rules affect its definition of equality, and the `==` equality operator performs type conversions as described in §3.9.1. (In practice, however, the `==` equality operator is deprecated in favor of the strict equality operator `===`, which does no type conversions. See §4.9.1 for more about both operators.)

Constants and variables allow you to use names to refer to values in your programs. Constants are declared with `const` and variables are declared with `let` (or with `var` in older JavaScript code). JavaScript constants and variables are *untyped*: declarations do not specify what kind of values will be assigned. Variable declaration and assignment are covered in §3.10.

As you can see from this long introduction, this is a wide-ranging chapter that explains many fundamental details about how data is represented and manipulated in JavaScript. We'll begin by diving right in to the details of JavaScript numbers and text.

3.2 Numbers

JavaScript's primary numeric type, Number, is used to represent integers and to approximate real numbers. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard,[1] which means it can represent numbers as large as $\pm 1.7976931348623157 \times 10^{308}$ and as small as $\pm 5 \times 10^{-324}$.

The JavaScript number format allows you to exactly represent all integers between −9,007,199,254,740,992 (-2^{53}) and 9,007,199,254,740,992 (2^{53}), inclusive. If you use integer values larger than this, you may lose precision in the trailing digits. Note, however, that certain operations in JavaScript (such as array indexing and the bitwise operators described in Chapter 4) are performed with 32-bit integers. If you need to exactly represent larger integers, see §3.2.5.

When a number appears directly in a JavaScript program, it's called a *numeric literal*. JavaScript supports numeric literals in several formats, as described in the following sections. Note that any numeric literal can be preceded by a minus sign (-) to make the number negative.

3.2.1 Integer Literals

In a JavaScript program, a base-10 integer is written as a sequence of digits. For example:

```
0
3
10000000
```

In addition to base-10 integer literals, JavaScript recognizes hexadecimal (base-16) values. A hexadecimal literal begins with `0x` or `0X`, followed by a string of hexadecimal digits. A hexadecimal digit is one of the digits 0 through 9 or the letters a (or A) through f (or F), which represent values 10 through 15. Here are examples of hexadecimal integer literals:

```
0xff       // => 255: (15*16 + 15)
0xBADCAFE  // => 195939070
```

In ES6 and later, you can also express integers in binary (base 2) or octal (base 8) using the prefixes `0b` and `0o` (or `0B` and `0O`) instead of `0x`:

```
0b10101  // => 21:  (1*16 + 0*8 + 1*4 + 0*2 + 1*1)
0o377    // => 255: (3*64 + 7*8 + 7*1)
```

3.2.2 Floating-Point Literals

Floating-point literals can have a decimal point; they use the traditional syntax for real numbers. A real value is represented as the integral part of the number, followed by a decimal point and the fractional part of the number.

Floating-point literals may also be represented using exponential notation: a real number followed by the letter e (or E), followed by an optional plus or minus sign, followed by an integer exponent. This notation represents the real number multiplied by 10 to the power of the exponent.

More succinctly, the syntax is:

```
[digits][.digits][(E|e)[(+|-)]digits]
```

For example:

```
3.14
2345.6789
.333333333333333333
6.02e23        // 6.02 × 10²³
1.4738223E-32  // 1.4738223 × 10⁻³²
```

Separators in Numeric Literals

You can use underscores within numeric literals to break long literals up into chunks that are easier to read:

```
let billion = 1_000_000_000;   // Underscore as a thousands separator.
let bytes = 0x89_AB_CD_EF;     // As a bytes separator.
let bits = 0b0001_1101_0111;   // As a nibble separator.
let fraction = 0.123_456_789;  // Works in the fractional part, too.
```

At the time of this writing in early 2020, underscores in numeric literals are not yet formally standardized as part of JavaScript. But they are in the advanced stages of the standardization process and are implemented by all major browsers and by Node.

3.2.3 Arithmetic in JavaScript

JavaScript programs work with numbers using the arithmetic operators . that the language provides. These include `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `%` for modulo (remainder after division). ES2016 adds `**` for exponentiation. Full details on these and other operators can be found in Chapter 4.

In addition to these basic arithmetic operators, JavaScript supports more complex mathematical operations through a set of functions and constants defined as properties of the `Math` object:

```
Math.pow(2,53)           // => 9007199254740992: 2 to the power 53
Math.round(.6)           // => 1.0: round to the nearest integer
Math.ceil(.6)            // => 1.0: round up to an integer
Math.floor(.6)           // => 0.0: round down to an integer
Math.abs(-5)             // => 5: absolute value
Math.max(x,y,z)          // Return the largest argument
Math.min(x,y,z)          // Return the smallest argument
Math.random()            // Pseudo-random number x where 0 <= x < 1.0
Math.PI                  // π: circumference of a circle / diameter
Math.E                   // e: The base of the natural logarithm
Math.sqrt(3)             // => 3**0.5: the square root of 3
Math.pow(3, 1/3)         // => 3**(1/3): the cube root of 3
Math.sin(0)              // Trigonometry: also Math.cos, Math.atan, etc.
Math.log(10)             // Natural logarithm of 10
Math.log(100)/Math.LN10  // Base 10 logarithm of 100
Math.log(512)/Math.LN2   // Base 2 logarithm of 512
Math.exp(3)              // Math.E cubed
```

ES6 defines more functions on the `Math` object:

```
Math.cbrt(27)    // => 3: cube root
Math.hypot(3, 4) // => 5: square root of sum of squares of all arguments
Math.log10(100)  // => 2: Base-10 logarithm
Math.log2(1024)  // => 10: Base-2 logarithm
Math.log1p(x)    // Natural log of (1+x); accurate for very small x
Math.expm1(x)    // Math.exp(x)-1; the inverse of Math.log1p()
Math.sign(x)     // -1, 0, or 1 for arguments <, ==, or > 0
Math.imul(2,3)   // => 6: optimized multiplication of 32-bit integers
Math.clz32(0xf)  // => 28: number of leading zero bits in a 32-bit integer
Math.trunc(3.9)  // => 3: convert to an integer by truncating fractional part
Math.fround(x)   // Round to nearest 32-bit float number
Math.sinh(x)     // Hyperbolic sine. Also Math.cosh(), Math.tanh()
Math.asinh(x)    // Hyperbolic arcsine. Also Math.acosh(), Math.atanh()
```

Arithmetic in JavaScript does not raise errors in cases of overflow, underflow, or division by zero. When the result of a numeric operation is larger than the largest representable number (overflow), the result is a special infinity value, `Infinity`. Similarly, when the absolute value of a negative value becomes larger than the absolute value of the largest representable negative number, the result is negative infinity, `-Infinity`. The infinite values behave as you would expect: adding, subtracting, multiplying, or dividing them by anything results in an infinite value (possibly with the sign reversed).

Underflow occurs when the result of a numeric operation is closer to zero than the smallest representable number. In this case, JavaScript returns 0. If underflow occurs from a negative number, JavaScript returns a special value known as “negative zero.” This value is almost completely indistinguishable from regular zero and JavaScript programmers rarely need to detect it.

Division by zero is not an error in JavaScript: it simply returns infinity or negative infinity. There is one exception, however: zero divided by zero does not have a well-defined value, and the result of this operation is the special not-a-number value, `NaN`. `NaN` also arises if you attempt to divide infinity by infinity, take the square root of a negative number, or use arithmetic operators with non-numeric operands that cannot be converted to numbers.

JavaScript predefines global constants `Infinity` and `NaN` to hold the positive infinity and not-a-number value, and these values are also available as properties of the `Number` object:

```
Infinity                    // A positive number too big to represent
Number.POSITIVE_INFINITY    // Same value
1/0                         // => Infinity
Number.MAX_VALUE * 2        // => Infinity; overflow

-Infinity                   // A negative number too big to represent
Number.NEGATIVE_INFINITY    // The same value
-1/0                        // => -Infinity
-Number.MAX_VALUE * 2       // => -Infinity

NaN                         // The not-a-number value
Number.NaN                  // The same value, written another way
0/0                         // => NaN
Infinity/Infinity           // => NaN

Number.MIN_VALUE/2          // => 0: underflow
-Number.MIN_VALUE/2         // => -0: negative zero
-1/Infinity                 // -> -0: also negative 0
-0

// The following Number properties are defined in ES6
Number.parseInt()       // Same as the global parseInt() function
Number.parseFloat()     // Same as the global parseFloat() function
Number.isNaN(x)         // Is x the NaN value?
Number.isFinite(x)      // Is x a number and finite?
Number.isInteger(x)     // Is x an integer?
Number.isSafeInteger(x) // Is x an integer -(2**53) < x < 2**53?
Number.MIN_SAFE_INTEGER // => -(2**53 - 1)
Number.MAX_SAFE_INTEGER // => 2**53 - 1
Number.EPSILON          // => 2**-52: smallest difference between numbers
```

The not-a-number value has one unusual feature in JavaScript: it does not compare equal to any other value, including itself. This means that you can't write `x === NaN` to determine whether the value of a variable `x` is `NaN`. Instead, you must write `x != x` or `Number.isNaN(x)`. Those expressions will be true if, and only if, `x` has the same value as the global constant `NaN`.

The global function `isNaN()` is similar to `Number.isNaN()`. It returns `true` if its argument is `NaN`, or if that argument is a non-numeric value that cannot be converted to a number. The related function `Number.isFinite()` returns `true` if its argument is a number other than `NaN`, `Infinity`, or `-Infinity`. The global `isFinite()` function returns `true` if its argument is, or can be converted to, a finite number.

The negative zero value is also somewhat unusual. It compares equal (even using JavaScript's strict equality test) to positive zero, which means that the two values are almost indistinguishable, except when used as a divisor:

```
let zero = 0;         // Regular zero
let negz = -0;        // Negative zero
zero === negz         // => true: zero and negative zero are equal
1/zero === 1/negz     // => false: Infinity and -Infinity are not equal
```

3.2.4 Binary Floating-Point and Rounding Errors

There are infinitely many real numbers, but only a finite number of them (18,437,736,874,454,810,627, to be exact) can be represented exactly by the JavaScript floating-point format. This means that when you're working with real numbers in JavaScript, the representation of the number will often be an approximation of the actual number.

The IEEE-754 floating-point representation used by JavaScript (and just about every other modern programming language) is a binary representation, which can exactly represent fractions like `1/2`, `1/8`, and `1/1024`. Unfortunately, the fractions we use most commonly (especially when performing financial calculations) are decimal fractions: `1/10`, `1/100`, and so on. Binary floating-point representations cannot exactly represent numbers as simple as `0.1`.

JavaScript numbers have plenty of precision and can approximate `0.1` very closely. But the fact that this number cannot be represented exactly can lead to problems. Consider this code:

```
let x = .3 - .2;    // thirty cents minus 20 cents
let y = .2 - .1;    // twenty cents minus 10 cents
x === y             // => false: the two values are not the same!
x === .1            // => false: .3-.2 is not equal to .1
y === .1            // => true: .2-.1 is equal to .1
```

Because of rounding error, the difference between the approximations of .3 and .2 is not exactly the same as the difference between the approximations of .2 and .1. It is important to understand that this problem is not specific to JavaScript: it affects any programming language that uses binary floating-point numbers. Also, note that the values `x` and `y` in the code shown here are *very* close to each other and to the correct value. The computed values are adequate for almost any purpose; the problem only arises when we attempt to compare values for equality.

If these floating-point approximations are problematic for your programs, consider using scaled integers. For example, you might manipulate monetary values as integer cents rather than fractional dollars.

3.2.5 Arbitrary Precision Integers with BigInt

One of the newest features of JavaScript, defined in ES2020, is a new numeric type known as BigInt. As of early 2020, it has been implemented in Chrome, Firefox, Edge, and Node, and there is an implementation in progress in Safari. As the name implies, BigInt is a numeric type whose values are integers. The type was added to JavaScript mainly to allow the representation of 64-bit integers, which are required for compatibility with many other programming languages and APIs. But BigInt values can have thousands or even millions of digits, should you have need to work with numbers that large. (Note, however, that BigInt implementations are not suitable for cryptography because they do not attempt to prevent timing attacks.)

BigInt literals are written as a string of digits followed by a lowercase letter `n`. By default, the are in base 10, but you can use the `0b`, `0o`, and `0x` prefixes for binary, octal, and hexadecimal BigInts:

```
1234n                // A not-so-big BigInt literal
0b111111n            // A binary BigInt
0o7777n              // An octal BigInt
0x8000000000000000n  // => 2n**63n: A 64-bit integer
```

You can use `BigInt()` as a function for converting regular JavaScript numbers or strings to BigInt values:

```
BigInt(Number.MAX_SAFE_INTEGER)     // => 9007199254740991n
let string = "1" + "0".repeat(100); // 1 followed by 100 zeros.
BigInt(string)                      // => 10n**100n: one googol
```

Arithmetic with BigInt values works like arithmetic with regular JavaScript numbers, except that division drops any remainder and rounds down (toward zero):

```
1000n + 2000n  // => 3000n
3000n - 2000n  // => 1000n
2000n * 3000n  // => 6000000n
3000n / 997n   // => 3n: the quotient is 3
3000n % 997n   // => 9n: and the remainder is 9
(2n ** 131071n) - 1n  // A Mersenne prime with 39457 decimal digits
```

Although the standard `+`, `-`, `*`, `/`, `%`, and `**` operators work with BigInt, it is important to understand that you may not mix operands of type BigInt with regular number operands. This may seem confusing at first, but there is a good reason for it. If one numeric type was more general than the other, it would be easy to define arithmetic on mixed operands to simply return a value of the more general type. But neither type is more general than the other: BigInt can represent extraordinarily large values, making it more general than regular numbers. But BigInt can only represent integers, making the regular JavaScript number type more general. There is no way around this problem, so JavaScript sidesteps it by simply not allowing mixed operands to the arithmetic operators.

Comparison operators, by contrast, do work with mixed numeric types (but see §3.9.1 for more about the difference between `==` and `===`):

```
1 < 2n     // => true
2 > 1n     // => true
```

```
0 == 0n    // => true
0 === 0n   // => false: the === checks for type equality as well
```

The bitwise operators (described in §4.8.3) generally work with BigInt operands. None of the functions of the `Math` object accept BigInt operands, however.

3.2.6 Dates and Times

JavaScript defines a simple Date class for representing and manipulating the numbers that represent dates and times. JavaScript Dates are objects, but they also have a numeric representation as a *timestamp* that specifies the number of elapsed milliseconds since January 1, 1970:

```
let timestamp = Date.now();  // The current time as a timestamp (a number).
let now = new Date();        // The current time as a Date object.
let ms = now.getTime();      // Convert to a millisecond timestamp.
let iso = now.toISOString(); // Convert to a string in standard format.
```

The Date class and its methods are covered in detail in §11.4. But we will see Date objects again in §3.9.3 when we examine the details of JavaScript type conversions.

3.3 Text

The JavaScript type for representing text is the *string*. A string is an immutable ordered sequence of 16-bit values, each of which typically represents a Unicode character. The *length* of a string is the number of 16-bit values it contains. JavaScript's strings (and its arrays) use zero-based indexing: the first 16-bit value is at position 0, the second at position 1, and so on. The *empty string* is the string of length 0. JavaScript does not have a special type that represents a single element of a string. To represent a single 16-bit value, simply use a string that has a length of 1.

Characters, Codepoints, and JavaScript Strings

JavaScript uses the UTF-16 encoding of the Unicode character set, and JavaScript strings are sequences of unsigned 16-bit values. The most commonly used Unicode characters (those from the "basic multilingual plane") have codepoints that fit in 16 bits and can be represented by one element of a string. Unicode characters whose codepoints do not fit in 16 bits are encoded using the rules of UTF-16 as a sequence (known as a "surrogate pair") of two 16-bit values. This means that a JavaScript string of length 2 (two 16-bit values) might represent only a single Unicode character:

```
let euro = "€";
let love = "❤";
euro.length   // => 1: this character has one 16-bit element
love.length   // => 2: UTF-16 encoding of ❤ is "\ud83d\udc99"
```

Most string-manipulation methods defined by JavaScript operate on 16-bit values, not characters. They do not treat surrogate pairs specially, they perform no normalization of the string, and don't even ensure that a string is well-formed UTF-16.

In ES6, however, strings are *iterable*, and if you use the `for/of` loop or `...` operator with a string, it will iterate the actual characters of the string, not the 16-bit values.

3.3.1 String Literals

To include a string in a JavaScript program, simply enclose the characters of the string within a matched pair of single or double quotes or backticks (`'` or `"` or `` ` ``). Double-quote characters and backticks may be contained within strings delimited by single-quote characters, and similarly for strings delimited by double quotes and backticks. Here are examples of string literals:

```
""  // The empty string: it has zero characters
'testing'
"3.14"
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"τ is the ratio of a circle's circumference to its radius"
`"She said 'hi'", he said.`
```

Strings delimited with backticks are a feature of ES6, and allow JavaScript expressions to be embedded within (or *interpolated* into) the string literal. This expression interpolation syntax is covered in §3.3.4.

The original versions of JavaScript required string literals to be written on a single line, and it is common to see JavaScript code that creates long strings by concatenating single-line strings with the `+` operator. As of ES5, however, you can break a string literal across multiple lines by ending each line but the last with a backslash (\). Neither the backslash nor the line terminator that follow it are part of the string literal. If you need to include a newline character in a single-quoted or double-quoted string literal, use the character sequence `\n` (documented in the next section). The ES6 backtick syntax allows strings to be broken across multiple lines, and in this case, the line terminators are part of the string literal:

```
// A string representing 2 lines written on one line:
'two\nlines'

// A one-line string written on 3 lines:
"one\
 long\
 line"

// A two-line string written on two lines:
`the newline character at the end of this line
is included literally in this string`
```

Note that when you use single quotes to delimit your strings, you must be careful with English contractions and possessives, such as *can't* and *O'Reilly's*. Since the apostrophe is the same as the single-quote character, you must use the backslash character (\) to "escape" any apostrophes that appear in single-quoted strings (escapes are explained in the next section).

In client-side JavaScript programming, JavaScript code may contain strings of HTML code, and HTML code may contain strings of JavaScript code. Like JavaScript, HTML uses either single or double quotes to delimit its strings. Thus, when combining JavaScript and HTML, it is a good idea to use one style of quotes for JavaScript and the other style for HTML. In the following example, the string "Thank you" is single-quoted within a JavaScript expression, which is then double-quoted within an HTML event-handler attribute:

```
<button onclick="alert('Thank you')">Click Me</button>
```

3.3.2 Escape Sequences in String Literals

The backslash character (\) has a special purpose in JavaScript strings. Combined with the character that follows it, it represents a character that is not otherwise representable within the string. For example, `\n` is an *escape sequence* that represents a newline character.

Another example, mentioned earlier, is the `\'` escape, which represents the single quote (or apostrophe) character. This escape sequence is useful when you need to include an apostrophe in a string literal that is contained within single quotes. You can see why these are called escape sequences: the backslash allows you to escape from the usual interpretation of the single-quote character. Instead of using it to mark the end of the string, you use it as an apostrophe:

```
'You\'re right, it can\'t be a quote'
```

Table 3-1 lists the JavaScript escape sequences and the characters they represent. Three escape sequences are generic and can be used to represent any character by specifying its Unicode character code as a hexadecimal number. For example, the sequence `\xA9` represents the copyright symbol, which has the Unicode encoding given by the hexadecimal number A9. Similarly, the `\u` escape represents an arbitrary Unicode character specified by four hexadecimal digits or one to five digits when the digits are enclosed in curly braces: `\u03c0` represents the character π, for example, and `\u{1f600}` represents the "grinning face" emoji.

Table 3-1. JavaScript escape sequences

Sequence	Character represented
`\0`	The NUL character (`\u0000`)
`\b`	Backspace (`\u0008`)
`\t`	Horizontal tab (`\u0009`)
`\n`	Newline (`\u000A`)
`\v`	Vertical tab (`\u000B`)
`\f`	Form feed (`\u000C`)
`\r`	Carriage return (`\u000D`)
`\"`	Double quote (`\u0022`)
`\'`	Apostrophe or single quote (`\u0027`)
`\\`	Backslash (`\u005C`)
`\x`*nn*	The Unicode character specified by the two hexadecimal digits *nn*
`\u`*nnnn*	The Unicode character specified by the four hexadecimal digits *nnnn*
`\u{`*n*`}`	The Unicode character specified by the codepoint *n*, where *n* is one to six hexadecimal digits between 0 and 10FFFF (ES6)

If the `\` character precedes any character other than those shown in Table 3-1, the backslash is simply ignored (although future versions of the language may, of course, define new escape sequences). For example, `\#` is the same as `#`. Finally, as noted earlier, ES5 allows a backslash before a line break to break a string literal across multiple lines.

3.3.3 Working with Strings

One of the built-in features of JavaScript is the ability to *concatenate* strings. If you use the `+` operator with numbers, it adds them. But if you use this operator on strings, it joins them by appending the second to the first. For example:

```
let msg = "Hello, " + "world";   // Produces the string "Hello, world"
let greeting = "Welcome to my blog," + " " + name;
```

Strings can be compared with the standard `===` equality and `!==` inequality operators: two strings are equal if and only if they consist of exactly the same sequence of 16-bit values. Strings can also be compared with the `<`, `<=`, `>`, and `>=` operators. String comparison is done simply by comparing the 16-bit values. (For more robust locale-aware string comparison and sorting, see §11.7.3.)

To determine the length of a string—the number of 16-bit values it contains—use the `length` property of the string:

```
s.length
```

In addition to this `length` property, JavaScript provides a rich API for working with strings:

```
let s = "Hello, world"; // Start with some text.

// Obtaining portions of a string
s.substring(1,4)        // => "ell": the 2nd, 3rd, and 4th characters.
s.slice(1,4)            // => "ell": same thing
s.slice(-3)             // => "rld": last 3 characters
s.split(", ")           // => ["Hello", "world"]: split at delimiter string

// Searching a string
s.indexOf("l")          // => 2: position of first letter l
s.indexOf("l", 3)       // => 3: position of first "l" at or after 3
s.indexOf("zz")         // => -1: s does not include the substring "zz"
s.lastIndexOf("l")      // => 10: position of last letter l

// Boolean searching functions in ES6 and later
s.startsWith("Hell")    // => true: the string starts with these
s.endsWith("!")         // => false: s does not end with that
s.includes("or")        // => true: s includes substring "or"

// Creating modified versions of a string
s.replace("llo", "ya")  // => "Heya, world"
s.toLowerCase()         // => "hello, world"
s.toUpperCase()         // => "HELLO, WORLD"
s.normalize()           // Unicode NFC normalization: ES6
s.normalize("NFD")      // NFD normalization. Also "NFKC", "NFKD"

// Inspecting individual (16-bit) characters of a string
s.charAt(0)             // => "H": the first character
s.charAt(s.length-1)    // => "d": the last character
s.charCodeAt(0)         // => 72: 16-bit number at the specified position
s.codePointAt(0)        // => 72: ES6, works for codepoints > 16 bits

// String padding functions in ES2017
"x".padStart(3)         // => "  x": add spaces on the left to a length of 3
"x".padEnd(3)           // => "x  ": add spaces on the right to a length of 3
"x".padStart(3, "*")    // => "**x": add stars on the left to a length of 3
"x".padEnd(3, "-")      // => "x--": add dashes on the right to a length of 3

// Space trimming functions. trim() is ES5; others ES2019
" test ".trim()         // => "test": remove spaces at start and end
" test ".trimStart()    // => "test ": remove spaces on left. Also trimLeft
" test ".trimEnd()      // => " test": remove spaces at right. Also trimRight

// Miscellaneous string methods
s.concat("!")           // => "Hello, world!": just use + operator instead
"<>".repeat(5)          // => "<><><><><>": concatenate n copies. ES6
```

Remember that strings are immutable in JavaScript. Methods like `replace()` and `toUpperCase()` return new strings: they do not modify the string on which they are invoked.

Strings can also be treated like read-only arrays, and you can access individual characters (16-bit values) from a string using square brackets instead of the `charAt()` method:

```
let s = "hello, world";
s[0]                  // => "h"
s[s.length-1]         // => "d"
```

3.3.4 Template Literals

In ES6 and later, string literals can be delimited with backticks:

```
let s = `hello world`;
```

This is more than just another string literal syntax, however, because these *template literals* can include arbitrary JavaScript expressions. The final value of a string literal in backticks is computed by evaluating any included expressions, converting the values of those expressions to strings and combining those computed strings with the literal characters within the backticks:

```
let name = "Bill";
let greeting = `Hello ${ name }.`;  // greeting == "Hello Bill."
```

Everything between the `${` and the matching `}` is interpreted as a JavaScript expression. Everything outside the curly braces is normal string literal text. The expression inside the braces is evaluated and then converted to a string and inserted into the template, replacing the dollar sign, the curly braces, and everything in between them.

A template literal may include any number of expressions. It can use any of the escape characters that normal strings can, and it can span any number of lines, with no special escaping required. The following template literal includes four JavaScript expressions, a Unicode escape sequence, and at least four newlines (the expression values may include newlines as well):

```
let errorMessage = `\
\u2718 Test failure at ${filename}:${linenumber}:
${exception.message}
Stack trace:
${exception.stack}
`;
```

The backslash at the end of the first line here escapes the initial newline so that the resulting string begins with the Unicode ✘ character (`\u2718`) rather than a newline.

Tagged template literals

A powerful but less commonly used feature of template literals is that, if a function name (or "tag") comes right before the opening backtick, then the text and the values of the expressions within the template literal are passed to the function. The value of this "tagged template literal" is the return value of the function. This could be used, for example, to apply HTML or SQL escaping to the values before substituting them into the text.

ES6 has one built-in tag function: `String.raw()`. It returns the text within backticks without any processing of backslash escapes:

```
`\n`.length            // => 1: the string has a single newline character
String.raw`\n`.length  // => 2: a backslash character and the letter n
```

Note that even though the tag portion of a tagged template literal is a function, there are no parentheses used in its invocation. In this very specific case, the backtick characters replace the open and close parentheses.

The ability to define your own template tag functions is a powerful feature of JavaScript. These functions do not need to return strings, and they can be used like constructors, as if defining a new literal syntax for the language. We'll see an example in §14.5.

3.3.5 Pattern Matching

JavaScript defines a datatype known as a *regular expression* (or RegExp) for describing and matching patterns in strings of text. RegExps are not one of the fundamental datatypes in JavaScript, but they have a literal syntax like numbers and strings do, so they sometimes seem like they are fundamental. The grammar of regular expression literals is complex and the API they define is nontrivial. They are documented in detail in §11.3. Because RegExps are powerful and commonly used for text processing, however, this section provides a brief overview.

Text between a pair of slashes constitutes a regular expression literal. The second slash in the pair can also be followed by one or more letters, which modify the meaning of the pattern. For example:

```
/^HTML/;             // Match the letters H T M L at the start of a string
/[1-9][0-9]*/;       // Match a nonzero digit, followed by any # of digits
/\bjavascript\b/i;   // Match "javascript" as a word, case-insensitive
```

RegExp objects define a number of useful methods, and strings also have methods that accept RegExp arguments. For example:

```
let text = "testing: 1, 2, 3";   // Sample text
let pattern = /\d+/g;            // Matches all instances of one or more digits
pattern.test(text)               // => true: a match exists
text.search(pattern)             // => 9: position of first match
text.match(pattern)              // => ["1", "2", "3"]: array of all matches
text.replace(pattern, "#")       // => "testing: #, #, #"
text.split(/\D+/)                // => ["","1","2","3"]: split on nondigits
```

3.4 Boolean Values

A boolean value represents truth or falsehood, on or off, yes or no. There are only two possible values of this type. The reserved words `true` and `false` evaluate to these two values.

Boolean values are generally the result of comparisons you make in your JavaScript programs. For example:

```
a === 4
```

This code tests to see whether the value of the variable `a` is equal to the number `4`. If it is, the result of this comparison is the boolean value `true`. If `a` is not equal to `4`, the result of the comparison is `false`.

Boolean values are commonly used in JavaScript control structures. For example, the `if/else` statement in JavaScript performs one action if a boolean value is `true` and another action if the value is `false`. You usually combine a comparison that creates a boolean value directly with a statement that uses it. The result looks like this:

```
if (a === 4) {
    b = b + 1;
} else {
    a = a + 1;
}
```

This code checks whether `a` equals `4`. If so, it adds `1` to `b`; otherwise, it adds `1` to `a`.

As we'll discuss in §3.9, any JavaScript value can be converted to a boolean value. The following values convert to, and therefore work like, `false`:

```
undefined
null
0
-0
NaN
""  // the empty string
```

All other values, including all objects (and arrays) convert to, and work like, `true`. `false`, and the six values that convert to it, are sometimes called *falsy* values, and all other values are called *truthy*. Any time JavaScript expects a boolean value, a falsy value works like `false` and a truthy value works like `true`.

As an example, suppose that the variable `o` either holds an object or the value `null`. You can test explicitly to see if `o` is non-null with an `if` statement like this:

```
if (o !== null) ...
```

The not-equal operator `!==` compares `o` to `null` and evaluates to either `true` or `false`. But you can omit the comparison and instead rely on the fact that `null` is falsy and objects are truthy:

```
if (o) ...
```

In the first case, the body of the `if` will be executed only if `o` is not `null`. The second case is less strict: it will execute the body of the `if` only if `o` is not `false` or any falsy value (such as `null` or `undefined`). Which `if` statement is appropriate for your program really depends on what values you expect to be assigned to `o`. If you need to distinguish `null` from `0` and `""`, then you should use an explicit comparison.

Boolean values have a `toString()` method that you can use to convert them to the strings "true" or "false", but they do not have any other useful methods. Despite the trivial API, there are three important boolean operators.

The `&&` operator performs the Boolean AND operation. It evaluates to a truthy value if and only if both of its operands are truthy; it evaluates to a falsy value otherwise. The `||` operator is the Boolean OR operation: it evaluates to a truthy value if either one (or both) of its operands is truthy and evaluates to a falsy value if both operands are falsy. Finally, the unary `!` operator performs the Boolean NOT operation: it evaluates to `true` if its operand is falsy and evaluates to `false` if its operand is truthy. For example:

```
if ((x === 0 && y === 0) || !(z === 0)) {
    // x and y are both zero or z is non-zero
}
```

Full details on these operators are in §4.10.

3.5 null and undefined

`null` is a language keyword that evaluates to a special value that is usually used to indicate the absence of a value. Using the `typeof` operator on `null` returns the string “object”, indicating that `null` can be thought of as a special object value that indicates “no object”. In practice, however, `null` is typically regarded as the sole member of its own type, and it can be used to indicate “no value” for numbers and strings as well as objects. Most programming languages have an equivalent to JavaScript’s `null`: you may be familiar with it as `NULL`, `nil`, or `None`.

JavaScript also has a second value that indicates absence of value. The `undefined` value represents a deeper kind of absence. It is the value of variables that have not been initialized and the value you get when you query the value of an object property or array element that does not exist. The `undefined` value is also the return value of functions that do not explicitly return a value and the value of function parameters for which no argument is passed. `undefined` is a predefined global constant (not a language keyword like `null`, though this is not an important distinction in practice) that is initialized to the `undefined` value. If you apply the `typeof` operator to the `undefined` value, it returns “undefined”, indicating that this value is the sole member of a special type.

Despite these differences, `null` and `undefined` both indicate an absence of value and can often be used interchangeably. The equality operator `==` considers them to be equal. (Use the strict equality operator `===` to distinguish them.) Both are falsy values: they behave like `false` when a boolean value is required. Neither `null` nor `undefined` have any properties or methods. In fact, using `.` or `[]` to access a property or method of these values causes a TypeError.

I consider `undefined` to represent a system-level, unexpected, or error-like absence of value and `null` to represent a program-level, normal, or expected absence of value. I avoid using `null` and `undefined` when I can, but if I need to assign one of these values to a variable or property or pass or return one of these values to or from a function, I usually use `null`. Some programmers strive to avoid `null` entirely and use `undefined` in its place wherever they can.

3.6 Symbols

Symbols were introduced in ES6 to serve as non-string property names. To understand Symbols, you need to know that JavaScript’s fundamental Object type is an unordered collection of properties, where each property has a name and a value. Property names are typically (and until ES6, were exclusively) strings. But in ES6 and later, Symbols can also serve this purpose:

```
let strname = "string name";      // A string to use as a property name
let symname = Symbol("propname"); // A Symbol to use as a property name
typeof strname                    // => "string": strname is a string
typeof symname                    // => "symbol": symname is a symbol
let o = {};                       // Create a new object
o[strname] = 1;                   // Define a property with a string name
o[symname] = 2;                   // Define a property with a Symbol name
o[strname]                        // => 1: access the string-named property
o[symname]                        // => 2: access the symbol-named property
```

The Symbol type does not have a literal syntax. To obtain a Symbol value, you call the `Symbol()` function. This function never returns the same value twice, even when called with the same argument. This means that if you call `Symbol()` to obtain a Symbol value, you can safely use that value as a property name to add a new property to an object and do not need to worry that you might be overwriting an existing property with the same name. Similarly, if you use symbolic property names and do not share those symbols, you can be confident that other modules of code in your program will not accidentally overwrite your properties.

In practice, Symbols serve as a language extension mechanism. When ES6 introduced the `for/of` loop (§5.4.4) and iterable objects (Chapter 12), it needed to define standard method that classes could implement to make themselves iterable. But standardizing any particular string name for this iterator method would have broken existing code, so a symbolic name was used instead. As we'll see in Chapter 12, `Symbol.iterator` is a Symbol value that can be used as a method name to make an object iterable.

The `Symbol()` function takes an optional string argument and returns a unique Symbol value. If you supply a string argument, that string will be included in the output of the Symbol's `toString()` method. Note, however, that calling `Symbol()` twice with the same string produces two completely different Symbol values.

```
let s = Symbol("sym_x");
s.toString()             // => "Symbol(sym_x)"
```

`toString()` is the only interesting method of Symbol instances. There are two other Symbol-related functions you should know about, however. Sometimes when using Symbols, you want to keep them private to your own code so you have a guarantee that your properties will never conflict with properties used by other code. Other times, however, you might want to define a Symbol value and share it widely with other code. This would be the case, for example, if you were defining some kind of extension that you wanted other code to be able to participate in, as with the `Symbol.iterator` mechanism described earlier.

To serve this latter use case, JavaScript defines a global Symbol registry. The `Symbol.for()` function takes a string argument and returns a Symbol value that is associated with the string you pass. If no Symbol is already associated with that string, then a new one is created and returned; otherwise, the already existing Symbol is returned. That is, the `Symbol.for()` function is completely different than the `Symbol()` function: `Symbol()` never returns the same value twice, but `Symbol.for()` always returns the same value when called with the same string. The string passed to `Symbol.for()` appears in the output of `toString()` for the returned Symbol, and it can also be retrieved by calling `Symbol.keyFor()` on the returned Symbol.

```
let s = Symbol.for("shared");
let t = Symbol.for("shared");
s === t          // => true
s.toString()     // => "Symbol(shared)"
Symbol.keyFor(t) // => "shared"
```

3.7 The Global Object

The preceding sections have explained JavaScript's primitive types and values. Object types—objects, arrays, and functions—are covered in chapters of their own later in this book. But there is one very important object value that we must cover now. The *global object* is a regular JavaScript object that serves a very important purpose: the properties of this object are the globally defined identifiers that are available to a JavaScript program. When the JavaScript interpreter starts (or whenever a web browser loads a new page), it creates a new global object and gives it an initial set of properties that define:

- Global constants like `undefined`, `Infinity`, and `NaN`
- Global functions like `isNaN()`, `parseInt()` (§3.9.2), and `eval()` (§4.12)
- Constructor functions like `Date()`, `RegExp()`, `String()`, `Object()`, and `Array()` (§3.9.2)
- Global objects like Math and JSON (§6.8)

The initial properties of the global object are not reserved words, but they deserve to be treated as if they are. This chapter has already described some of these global properties. Most of the others will be covered elsewhere in this book.

In Node, the global object has a property named `global` whose value is the global object itself, so you can always refer to the global object by the name `global` in Node programs.

In web browsers, the Window object serves as the global object for all JavaScript code contained in the browser window it represents. This global Window object has a self-referential `window` property that can be used to refer to the global object. The Window object defines the core global properties, but it also defines quite a few other globals that are specific to web browsers and client-side JavaScript. Web worker threads (§15.13) have a different global object than the Window with which they are associated. Code in a worker can refer to its global object as `self`.

ES2020 finally defines `globalThis` as the standard way to refer to the global object in any context. As of early 2020, this feature has been implemented by all modern browsers and by Node.

3.8 Immutable Primitive Values and Mutable Object References

There is a fundamental difference in JavaScript between primitive values (`undefined`, `null`, booleans, numbers, and strings) and objects (including arrays and functions). Primitives are immutable: there is no way to change (or "mutate") a primitive value. This is obvious for numbers and booleans—it doesn't even make sense to change the value of a number. It is not so obvious for strings, however. Since strings are like arrays of characters, you might expect to be able to alter the character at any specified index. In fact, JavaScript does not allow this, and all string methods that appear to return a modified string are, in fact, returning a new string value. For example:

```
let s = "hello";   // Start with some lowercase text
s.toUpperCase();   // Returns "HELLO", but doesn't alter s
s                  // => "hello": the original string has not changed
```

Primitives are also compared *by value*: two values are the same only if they have the same value. This sounds circular for numbers, booleans, `null`, and `undefined`: there is no other way that they could be compared. Again, however, it is not so obvious for strings. If two distinct string values are compared, JavaScript treats them as equal if, and only if, they have the same length and if the character at each index is the same.

Objects are different than primitives. First, they are *mutable*—their values can change:

```
let o = { x: 1 };  // Start with an object
o.x = 2;           // Mutate it by changing the value of a property
o.y = 3;           // Mutate it again by adding a new property

let a = [1,2,3];   // Arrays are also mutable
a[0] = 0;          // Change the value of an array element
a[3] = 4;          // Add a new array element
```

Objects are not compared by value: two distinct objects are not equal even if they have the same properties and values. And two distinct arrays are not equal even if they have the same elements in the same order:

```
let o = {x: 1}, p = {x: 1};  // Two objects with the same properties
o === p                      // => false: distinct objects are never equal
let a = [], b = [];          // Two distinct, empty arrays
a === b                      // => false: distinct arrays are never equal
```

Objects are sometimes called *reference types* to distinguish them from JavaScript's primitive types. Using this terminology, object values are *references*, and we say that objects are compared *by reference*: two object values are the same if and only if they *refer* to the same underlying object.

```
let a = [];    // The variable a refers to an empty array.
let b = a;     // Now b refers to the same array.
b[0] = 1;      // Mutate the array referred to by variable b.
a[0]           // => 1: the change is also visible through variable a.
a === b        // => true: a and b refer to the same object, so they are equal.
```

As you can see from this code, assigning an object (or array) to a variable simply assigns the reference: it does not create a new copy of the object. If you want to make a new copy of an object or array, you must explicitly copy the properties of the object or the elements of the array. This example demonstrates using a `for` loop (§5.4.3):

```
let a = ["a","b","c"];              // An array we want to copy
let b = [];                         // A distinct array we'll copy into
for(let i = 0; i < a.length; i++) { // For each index of a[]
    b[i] = a[i];                    // Copy an element of a into b
}
let c = Array.from(b);              // In ES6, copy arrays with Array.from()
```

Similarly, if we want to compare two distinct objects or arrays, we must compare their properties or elements. This code defines a function to compare two arrays:

```
function equalArrays(a, b) {
    if (a === b) return true;                // Identical arrays are equal
    if (a.length !== b.length) return false; // Different-size arrays not equal
    for(let i = 0; i < a.length; i++) {      // Loop through all elements
        if (a[i] !== b[i]) return false;     // If any differ, arrays not equal
    }
    return true;                             // Otherwise they are equal
}
```

3.9 Type Conversions

JavaScript is very flexible about the types of values it requires. We've seen this for booleans: when JavaScript expects a boolean value, you may supply a value of any type, and JavaScript will convert it as needed. Some values ("truthy" values) convert to `true` and others ("falsy" values) convert to `false`. The same is true for other types: if JavaScript wants a string, it will convert whatever value you give it to a string. If JavaScript wants a number, it will try to convert the value you give it to a number (or to `NaN` if it cannot perform a meaningful conversion).

Some examples:

```
10 + " objects"     // => "10 objects":  Number 10 converts to a string
"7" * "4"           // => 28: both strings convert to numbers
let n = 1 - "x";    // n == NaN; string "x" can't convert to a number
n + " objects"      // => "NaN objects": NaN converts to string "NaN"
```

Table 3-2 summarizes how values convert from one type to another in JavaScript. Bold entries in the table highlight conversions that you may find surprising. Empty cells indicate that no conversion is necessary and none is performed.

Table 3-2. JavaScript type conversions

Value	to String	to Number	to Boolean
`undefined`	`"undefined"`	`NaN`	`false`
`null`	`"null"`	**`0`**	`false`
`true`	`"true"`	**`1`**	
`false`	`"false"`	**`0`**	
`""` (empty string)		**`0`**	**`false`**
`"1.2"` (nonempty, numeric)		`1.2`	`true`
`"one"` (nonempty, non-numeric)		`NaN`	`true`
`0`	`"0"`		**`false`**
`-0`	`"0"`		**`false`**
`1` (finite, non-zero)	`"1"`		`true`
`Infinity`	`"Infinity"`		`true`
`-Infinity`	`"-Infinity"`		`true`
`NaN`	`"NaN"`		**`false`**
`{}` (any object)	*see §3.9.3*	*see §3.9.3*	`true`
`[]` (empty array)	`""`	**`0`**	`true`
`[9]` (one numeric element)	`"9"`	**`9`**	`true`
`['a']` (any other array)	*use join() method*	`NaN`	`true`
`function(){}` (any function)	*see §3.9.3*	`NaN`	`true`

The primitive-to-primitive conversions shown in the table are relatively straightforward. Conversion to boolean was already discussed in §3.4. Conversion to strings is well defined for all primitive values. Conversion to numbers is just a little trickier. Strings that can be parsed as numbers convert to those numbers. Leading and trailing spaces are allowed, but any leading or trailing nonspace characters that are not part of a numeric literal cause the string-to-number conversion to produce `NaN`. Some numeric conversions may seem surprising: `true` converts to 1, and `false` and the empty string convert to 0.

Object-to-primitive conversion is somewhat more complicated, and it is the subject of §3.9.3.

3.9.1 Conversions and Equality

JavaScript has two operators that test whether two values are equal. The "strict equality operator," `===`, does not consider its operands to be equal if they are not of the same type, and this is almost always the right operator to use when coding. But because JavaScript is so flexible with type conversions, it also defines the `==` operator with a flexible definition of equality. All of the following comparisons are true, for example:

```
null == undefined // => true: These two values are treated as equal.
"0" == 0          // => true: String converts to a number before comparing.
```

```
0 == false        // => true: Boolean converts to number before comparing.
"0" == false      // => true: Both operands convert to 0 before comparing!
```

§4.9.1 explains exactly what conversions are performed by the `==` operator in order to determine whether two values should be considered equal.

Keep in mind that convertibility of one value to another does not imply equality of those two values. If `undefined` is used where a boolean value is expected, for example, it will convert to `false`. But this does not mean that `undefined == false`. JavaScript operators and statements expect values of various types and perform conversions to those types. The `if` statement converts `undefined` to `false`, but the `==` operator never attempts to convert its operands to booleans.

3.9.2 Explicit Conversions

Although JavaScript performs many type conversions automatically, you may sometimes need to perform an explicit conversion, or you may prefer to make the conversions explicit to keep your code clearer.

The simplest way to perform an explicit type conversion is to use the `Boolean()`, `Number()`, and `String()` functions:

```
Number("3")    // => 3
String(false)  // => "false":  Or use false.toString()
Boolean([])    // => true
```

Any value other than `null` or `undefined` has a `toString()` method, and the result of this method is usually the same as that returned by the `String()` function.

As an aside, note that the `Boolean()`, `Number()`, and `String()` functions can also be invoked—with `new`—as constructor. If you use them this way, you'll get a "wrapper" object that behaves just like a primitive boolean, number, or string value. These wrapper objects are a historical leftover from the earliest days of JavaScript, and there is never really any good reason to use them.

Certain JavaScript operators perform implicit type conversions and are sometimes used explicitly for the purpose of type conversion. If one operand of the `+` operator is a string, it converts the other one to a string. The unary `+` operator converts its operand to a number. And the unary `!` operator converts its operand to a boolean and negates it. These facts lead to the following type conversion idioms that you may see in some code:

```
x + ""   // => String(x)
+x       // => Number(x)
x-0      // => Number(x)
!!x      // => Boolean(x): Note double !
```

Formatting and parsing numbers are common tasks in computer programs, and JavaScript has specialized functions and methods that provide more precise control over number-to-string and string-to-number conversions.

The `toString()` method defined by the Number class accepts an optional argument that specifies a radix, or base, for the conversion. If you do not specify the argument, the conversion is done in base 10. However, you can also convert numbers in other bases (between 2 and 36). For example:

```
let n = 17;
let binary = "0b" + n.toString(2);  // binary == "0b10001"
let octal = "0o" + n.toString(8);   // octal == "0o21"
let hex = "0x" + n.toString(16);    // hex == "0x11"
```

When working with financial or scientific data, you may want to convert numbers to strings in ways that give you control over the number of decimal places or the number of significant digits in the output, or you may want to control whether exponential notation is used. The Number class defines three methods for these kinds of number-to-string conversions. `toFixed()` converts a number to a string with a specified number of digits after the decimal point. It never uses exponential notation. `toExponential()` converts a number to a string using exponential notation, with one digit before the decimal point and a specified number of digits after the decimal point (which means that the number of significant digits is one larger than the value you specify). `toPrecision()` converts a number to a string with the number of significant digits you specify. It uses exponential notation if the number of significant digits is not large enough to display the entire integer portion of the number. Note that all three methods round the trailing digits or pad with zeros as appropriate. Consider the following examples:

```
let n = 123456.789;
n.toFixed(0)         // => "123457"
n.toFixed(2)         // => "123456.79"
n.toFixed(5)         // => "123456.78900"
n.toExponential(1)   // => "1.2e+5"
n.toExponential(3)   // => "1.235e+5"
n.toPrecision(4)     // => "1.235e+5"
n.toPrecision(7)     // => "123456.8"
n.toPrecision(10)    // => "123456.7890"
```

In addition to the number-formatting methods shown here, the Intl.NumberFormat class defines a more general, internationalized number-formatting method. See §11.7.1 for details.

If you pass a string to the `Number()` conversion function, it attempts to parse that string as an integer or floating-point literal. That function only works for base-10 integers and does not allow trailing characters that are not part of the literal. The `parseInt()` and `parseFloat()` functions (these are global functions, not methods of any class) are more flexible. `parseInt()` parses only integers, while `parseFloat()` parses both integers and floating-point numbers. If a string begins with "0x" or "0X", `parseInt()` interprets it as a hexadecimal number. Both `parseInt()` and `parseFloat()` skip leading whitespace, parse as many numeric characters as they can, and ignore anything that follows. If the first nonspace character is not part of a valid numeric literal, they return `NaN`:

```
parseInt("3 blind mice")     // => 3
parseFloat(" 3.14 meters")   // => 3.14
parseInt("-12.34")           // => -12
parseInt("0xFF")             // => 255
parseInt("0xff")             // => 255
parseInt("-0XFF")            // => -255
parseFloat(".1")             // => 0.1
parseInt("0.1")              // => 0
parseInt(".1")               // => NaN: integers can't start with "."
parseFloat("$72.47")         // => NaN: numbers can't start with "$"
```

`parseInt()` accepts an optional second argument specifying the radix (base) of the number to be parsed. Legal values are between 2 and 36. For example:

```
parseInt("11", 2)     // => 3: (1*2 + 1)
parseInt("ff", 16)    // => 255: (15*16 + 15)
parseInt("zz", 36)    // => 1295: (35*36 + 35)
parseInt("077", 8)    // => 63: (7*8 + 7)
parseInt("077", 10)   // => 77: (7*10 + 7)
```

3.9.3 Object to Primitive Conversions

The previous sections have explained how you can explicitly convert values of one type to another type and have explained JavaScript's implicit conversions of values from one primitive type to another primitive type. This section covers the complicated rules that JavaScript uses to convert objects to primitive values. It is long and obscure, and if this is your first reading of this chapter, you should feel free to skip ahead to §3.10.

One reason for the complexity of JavaScript's object-to-primitive conversions is that some types of objects have more than one primitive representation. Date objects, for example, can be represented as strings or as numeric timestamps. The JavaScript specification defines three fundamental algorithms for converting objects to primitive values:

prefer-string

> This algorithm returns a primitive value, preferring a string value, if a conversion to string is possible.

prefer-number

> This algorithm returns a primitive value, preferring a number, if such a conversion is possible.

no-preference

> This algorithm expresses no preference about what type of primitive value is desired, and classes can define their own conversions. Of the built-in JavaScript types, all except Date implement this algorithm as *prefer-number*. The Date class implements this algorithm as *prefer-string*.

The implementation of these object-to-primitive conversion algorithms is explained at the end of this section. First, however, we explain how the algorithms are used in JavaScript.

Object-to-boolean conversions

Object-to-boolean conversions are trivial: all objects convert to `true`. Notice that this conversion does not require the use of the object-to-primitive algorithms described, and that it literally applies to *all* objects, including empty arrays and even the wrapper object `new Boolean(false)`.

Object-to-string conversions

When an object needs to be converted to a string, JavaScript first converts it to a primitive using the *prefer-string* algorithm, then converts the resulting primitive value to a string, if necessary, following the rules in Table 3-2.

This kind of conversion happens, for example, if you pass an object to a built-in function that expects a string argument, if you call `String()` as a conversion function, and when you interpolate objects into template literals (§3.3.4).

Object-to-number conversions

When an object needs to be converted to a number, JavaScript first converts it to a primitive value using the *prefer-number* algorithm, then converts the resulting primitive value to a number, if necessary, following the rules in Table 3-2.

Built-in JavaScript functions and methods that expect numeric arguments convert object arguments to numbers in this way, and most (see the exceptions that follow) JavaScript operators that expect numeric operands convert objects to numbers in this way as well.

Special case operator conversions

Operators are covered in detail in Chapter 4. Here, we explain the special case operators that do not use the basic object-to-string and object-to-number conversions described earlier.

The `+` operator in JavaScript performs numeric addition and string concatenation. If either of its operands is an object, JavaScript converts them to primitive values using the *no-preference* algorithm. Once it has two primitive values, it checks their types. If either argument is a string, it converts the other to a string and concatenates the strings. Otherwise, it converts both arguments to numbers and adds them.

The `==` and `!=` operators perform equality and inequality testing in a loose way that allows type conversions. If one operand is an object and the other is a primitive value, these operators convert the object to primitive using the *no-preference* algorithm and then compare the two primitive values.

Finally, the relational operators `<`, `<=`, `>`, and `>=` compare the order of their operands and can be used to compare both numbers and strings. If either operand is an object, it is converted to a primitive value using the *prefer-number* algorithm. Note, however, that unlike the object-to-number conversion, the primitive values returned by the *prefer-number* conversion are not then converted to numbers.

Note that the numeric representation of Date objects is meaningfully comparable with `<` and `>`, but the string representation is not. For Date objects, the *no-preference* algorithm converts to a string, so the fact that JavaScript uses the *prefer-number* algorithm for these operators means that we can use them to compare the order of two Date objects.

The toString() and valueOf() methods

All objects inherit two conversion methods that are used by object-to-primitive conversions, and before we can explain the *prefer-string*, *prefer-number*, and *no-preference* conversion algorithms, we have to explain these two methods.

The first method is `toString()`, and its job is to return a string representation of the object. The default `toString()` method does not return a very interesting value (though we'll find it useful in §14.4.3):

```
({x: 1, y: 2}).toString()    // => "[object Object]"
```

Many classes define more specific versions of the `toString()` method. The `toString()` method of the Array class, for example, converts each array element to a string and joins the resulting strings together with commas in between. The `toString()` method of the Function class converts user-defined functions to strings of JavaScript source code. The Date class defines a `toString()` method that returns a human-readable (and JavaScript-parsable) date and time string. The RegExp class defines a `toString()` method that converts RegExp objects to a string that looks like a RegExp literal:

```
[1,2,3].toString()                  // => "1,2,3"
(function(x) { f(x); }).toString()  // => "function(x) { f(x); }"
/\d+/g.toString()                   // => "/\\d+/g"
let d = new Date(2020,0,1);
d.toString()  // => "Wed Jan 01 2020 00:00:00 GMT-0800 (Pacific Standard Time)"
```

The other object conversion function is called `valueOf()`. The job of this method is less well defined: it is supposed to convert an object to a primitive value that represents the object, if any such primitive value exists. Objects are compound values, and most objects cannot really be represented by a single primitive value, so the default `valueOf()` method simply returns the object itself rather than returning a primitive. Wrapper classes such as String, Number, and Boolean define `valueOf()` methods that simply return the wrapped primitive value. Arrays, functions, and regular expressions simply inherit the default method. Calling `valueOf()` for instances of these types simply returns the object itself. The Date class defines a `valueOf()` method that returns the date in its internal representation: the number of milliseconds since January 1, 1970:

```
let d = new Date(2010, 0, 1);   // January 1, 2010, (Pacific time)
d.valueOf()                     // => 1262332800000
```

Object-to-primitive conversion algorithms

With the `toString()` and `valueOf()` methods explained, we can now explain approximately how the three object-to-primitive algorithms work (the complete details are deferred until §14.4.7):

- The *prefer-string* algorithm first tries the `toString()` method. If the method is defined and returns a primitive value, then JavaScript uses that primitive value (even if it is not a string!). If `toString()` does not exist or if it returns an object, then JavaScript tries the `valueOf()` method. If that method exists and returns a primitive value, then JavaScript uses that value. Otherwise, the conversion fails with a TypeError.

- The *prefer-number* algorithm works like the *prefer-string* algorithm, except that it tries `valueOf()` first and `toString()` second.

- The *no-preference* algorithm depends on the class of the object being converted. If the object is a Date object, then JavaScript uses the *prefer-string* algorithm. For any other object, JavaScript uses the *prefer-number* algorithm.

The rules described here are true for all built-in JavaScript types and are the default rules for any classes you define yourself. §14.4.7 explains how you can define your own object-to-primitive conversion algorithms for the classes you define.

Before we leave this topic, it is worth noting that the details of the *prefer-number* conversion explain why empty arrays convert to the number 0 and single-element arrays can also convert to numbers:

```
Number([])    // => 0: this is unexpected!
Number([99])  // => 99: really?
```

The object-to-number conversion first converts the object to a primitive using the *prefer-number* algorithm, then converts the resulting primitive value to a number. The *prefer-number* algorithm tries `valueOf()` first and then falls back on `toString()`. But the Array class inherits the default `valueOf()` method, which does not return a primitive value. So when we try to convert an array to a number, we end up invoking the `toString()` method of the array. Empty arrays convert to the empty string. And the empty string converts to the number 0. An array with a single element converts to the same string that that one element does. If an array contains a single number, that number is converted to a string, and then back to a number.

3.10 Variable Declaration and Assignment

One of the most fundamental techniques of computer programming is the use of names—or *identifiers*—to represent values. Binding a name to a value gives us a way to refer to that value and use it in the programs we write. When we do this, we typically say that we are assigning a value to a *variable*. The term “variable” implies that new values can be assigned: that the value associated with the variable may vary as our program runs. If we permanently assign a value to a name, then we call that name a *constant* instead of a variable.

Before you can use a variable or constant in a JavaScript program, you must *declare* it. In ES6 and later, this is done with the `let` and `const` keywords, which we explain next. Prior to ES6, variables were declared with `var`, which is more idiosyncratic and is explained later on in this section.

3.10.1 Declarations with let and const

In modern JavaScript (ES6 and later), variables are declared with the `let` keyword, like this:

```
let i;
let sum;
```

You can also declare multiple variables in a single `let` statement:

```
let i, sum;
```

It is a good programming practice to assign an initial value to your variables when you declare them, when this is possible:

```
let message = "hello";
let i = 0, j = 0, k = 0;
let x = 2, y = x*x; // Initializers can use previously declared variables
```

If you don’t specify an initial value for a variable with the `let` statement, the variable is declared, but its value is `undefined` until your code assigns a value to it.

To declare a constant instead of a variable, use `const` instead of `let`. `const` works just like `let` except that you must initialize the constant when you declare it:

```
const H0 = 74;         // Hubble constant (km/s/Mpc)
const C = 299792.458;  // Speed of light in a vacuum (km/s)
const AU = 1.496E8;    // Astronomical Unit: distance to the sun (km)
```

As the name implies, constants cannot have their values changed, and any attempt to do so causes a TypeError to be thrown.

It is a common (but not universal) convention to declare constants using names with all capital letters such as `H0` or `HTTP_NOT_FOUND` as a way to distinguish them from variables.

When to Use const

There are two schools of thought about the use of the `const` keyword. One approach is to use `const` only for values that are fundamentally unchanging, like the physical constants shown, or program version numbers, or byte sequences used to identify file types, for example. Another approach recognizes that many of the so-called variables in our program don't actually ever change as our program runs. In this approach, we declare everything with `const`, and then if we find that we do actually want to allow the value to vary, we switch the declaration to `let`. This may help prevent bugs by ruling out accidental changes to variables that we did not intend.

In one approach, we use `const` only for values that *must not* change. In the other, we use `const` for any value that does not happen to change. I prefer the former approach in my own code.

In Chapter 5, we'll learn about the `for`, `for/in`, and `for/of` loop statements in JavaScript. Each of these loops includes a loop variable that gets a new value assigned to it on each iteration of the loop. JavaScript allows us to declare the loop variable as part of the loop syntax itself, and this is another common way to use `let`:

```
for(let i = 0, len = data.length; i < len; i++) console.log(data[i]);
for(let datum of data) console.log(datum);
for(let property in object) console.log(property);
```

It may seem surprising, but you can also use `const` to declare the loop "variables" for `for/in` and `for/of` loops, as long as the body of the loop does not reassign a new value. In this case, the `const` declaration is just saying that the value is constant for the duration of one loop iteration:

```
for(const datum of data) console.log(datum);
for(const property in object) console.log(property);
```

Variable and constant scope

The *scope* of a variable is the region of your program source code in which it is defined. Variables and constants declared with `let` and `const` are *block scoped*. This means that they are only defined within the block of code in which the `let` or `const` statement appears. JavaScript class and function definitions are blocks, and so are the bodies of `if/else` statements, `while` loops, `for` loops, and so on. Roughly speaking, if a variable or constant is declared within a set of curly braces, then those curly braces delimit the region of code in which the variable or constant is defined (though of course it is not legal to reference a variable or constant from lines of code that execute before the `let` or `const` statement that declares the variable). Variables and constants declared as part of a `for`, `for/in`, or `for/of` loop have the loop body as their scope, even though they technically appear outside of the curly braces.

When a declaration appears at the top level, outside of any code blocks, we say it is a *global* variable or constant and has global scope. In Node and in client-side JavaScript modules (see Chapter 10), the scope of a global variable is the file that it is defined in. In traditional client-side JavaScript, however, the scope of a global variable is the HTML document in which it is defined. That is: if one `<script>` declares a global variable or constant, that variable or constant is defined in all of the `<script>` elements in that document (or at least all of the scripts that execute after the `let` or `const` statement executes).

Repeated declarations

It is a syntax error to use the same name with more than one `let` or `const` declaration in the same scope. It is legal (though a practice best avoided) to declare a new variable with the same name in a nested scope:

```
const x = 1;        // Declare x as a global constant
if (x === 1) {
    let x = 2;      // Inside a block x can refer to a different value
    console.log(x); // Prints 2
}
console.log(x);     // Prints 1: we're back in the global scope now
let x = 3;          // ERROR! Syntax error trying to re-declare x
```

Declarations and types

If you're used to statically typed languages such as C or Java, you may think that the primary purpose of variable declarations is to specify the type of values that may be assigned to a variable. But, as you have seen, there is no type associated with JavaScript's variable declarations.[2] A JavaScript variable can hold a value of any type. For example, it is perfectly legal (but generally poor programming style) in JavaScript to assign a number to a variable and then later assign a string to that variable:

```
let i = 10;
i = "ten";
```

3.10.2 Variable Declarations with var

In versions of JavaScript before ES6, the only way to declare a variable is with the `var` keyword, and there is no way to declare constants. The syntax of `var` is just like the syntax of `let`:

```
var x;
var data = [], count = data.length;
for(var i = 0; i < count; i++) console.log(data[i]);
```

Although `var` and `let` have the same syntax, there are important differences in the way they work:

- Variables declared with `var` do not have block scope. Instead, they are scoped to the body of the containing function no matter how deeply nested they are inside that function.

- If you use `var` outside of a function body, it declares a global variable. But global variables declared with `var` differ from globals declared with `let` in an important way. Globals declared with `var` are implemented as properties of the global object (§3.7). The global object can be referenced as `globalThis`. So if you write `var x = 2;` outside of a function, it is like you wrote `globalThis.x = 2;`. Note however, that the analogy is not perfect: the properties created with global `var` declarations cannot be deleted with the `delete` operator (§4.13.4). Global variables and constants declared with `let` and `const` are not properties of the global object.

- Unlike variables declared with `let`, it is legal to declare the same variable multiple times with `var`. And because `var` variables have function scope instead of block scope, it is actually common to do this kind of redeclaration. The variable `i` is frequently used for integer values, and especially as the index variable of `for` loops. In a function with multiple `for` loops, it is typical for each one to begin `for(var i = 0; ....` Because `var` does not scope these variables to the loop body, each of these loops is (harmlessly) re-declaring and re-initializing the same variable.

- One of the most unusual features of `var` declarations is known as *hoisting*. When a variable is declared with `var`, the declaration is lifted up (or "hoisted") to the top of the enclosing function. The initialization of the variable remains where you wrote it, but the definition of the variable moves to the top of the function. So variables declared with `var` can be used, without error, anywhere in the enclosing function. If the initialization code has not run yet, then the value of the variable may be `undefined`, but you won't get an error if you use the variable before it is initialized. (This can be a source of bugs and is one of the important misfeatures that `let` corrects: if you declare a variable with `let` but attempt to use it before the `let` statement runs, you will get an actual error instead of just seeing an `undefined` value.)

Using Undeclared Variables

In strict mode (§5.6.3), if you attempt to use an undeclared variable, you'll get a reference error when you run your code. Outside of strict mode, however, if you assign a value to a name that has not been declared with `let`, `const`, or `var`, you'll end up creating a new global variable. It will be a global no matter now deeply nested within functions and blocks your code is, which is almost certainly not what you want, is bug-prone, and is one of the best reasons for using strict mode!

Global variables created in this accidental way are like global variables declared with `var`: they define properties of the global object. But unlike the properties defined by proper `var` declarations, these properties *can* be deleted with the `delete` operator (§4.13.4).

3.10.3 Destructuring Assignment

ES6 implements a kind of compound declaration and assignment syntax known as *destructuring assignment*. In a destructuring assignment, the value on the righthand side of the equals sign is an array or object (a "structured" value), and the lefthand side specifies one or more variable names using a syntax that mimics array and object literal syntax. When a destructuring assignment occurs, one or more values are extracted ("destructured") from the value on the right and stored into the variables named on the left. Destructuring assignment is perhaps most commonly used to initialize variables as part of a `const`, `let`, or `var` declaration statement, but it can also be done in regular assignment expressions (with variables that have already been declared). And, as we'll see in §8.3.5, destructuring can also be used when defining the parameters to a function.

Here are simple destructuring assignments using arrays of values:

```
let [x,y] = [1,2];  // Same as let x=1, y=2
[x,y] = [x+1,y+1];  // Same as x = x + 1, y = y + 1
[x,y] = [y,x];      // Swap the value of the two variables
[x,y]               // => [3,2]: the incremented and swapped values
```

Notice how destructuring assignment makes it easy to work with functions that return arrays of values:

```
// Convert [x,y] coordinates to [r,theta] polar coordinates
function toPolar(x, y) {
    return [Math.sqrt(x*x+y*y), Math.atan2(y,x)];
}
```

```
// Convert polar to Cartesian coordinates
function toCartesian(r, theta) {
    return [r*Math.cos(theta), r*Math.sin(theta)];
}

let [r,theta] = toPolar(1.0, 1.0);  // r == Math.sqrt(2); theta == Math.PI/4
let [x,y] = toCartesian(r,theta);   // [x, y] == [1.0, 1,0]
```

We saw that variables and constants can be declared as part of JavaScript's various `for` loops. It is possible to use variable destructuring in this context as well. Here is a code that loops over the name/value pairs of all properties of an object and uses destructuring assignment to convert those pairs from two-element arrays into individual variables:

```
let o = { x: 1, y: 2 }; // The object we'll loop over
for(const [name, value] of Object.entries(o)) {
    console.log(name, value); // Prints "x 1" and "y 2"
}
```

The number of variables on the left of a destructuring assignment does not have to match the number of array elements on the right. Extra variables on the left are set to `undefined`, and extra values on the right are ignored. The list of variables on the left can include extra commas to skip certain values on the right:

```
let [x,y] = [1];     // x == 1; y == undefined
[x,y] = [1,2,3];     // x == 1; y == 2
[,x,,y] = [1,2,3,4]; // x == 2; y == 4
```

If you want to collect all unused or remaining values into a single variable when destructuring an array, use three dots (`...`) before the last variable name on the left-hand side:

```
let [x, ...y] = [1,2,3,4];  // y == [2,3,4]
```

We'll see three dots used this way again in §8.3.2, where they are used to indicate that all remaining function arguments should be collected into a single array.

Destructuring assignment can be used with nested arrays. In this case, the lefthand side of the assignment should look like a nested array literal:

```
let [a, [b, c]] = [1, [2,2.5], 3]; // a == 1; b == 2; c == 2.5
```

A powerful feature of array destructuring is that it does not actually require an array! You can use any *iterable* object (Chapter 12) on the righthand side of the assignment; any object that can be used with a `for/of` loop (§5.4.4) can also be destructured:

```
let [first, ...rest] = "Hello"; // first == "H"; rest == ["e","l","l","o"]
```

Destructuring assignment can also be performed when the righthand side is an object value. In this case, the lefthand side of the assignment looks something like an object literal: a comma-separated list of variable names within curly braces:

```
let transparent = {r: 0.0, g: 0.0, b: 0.0, a: 1.0}; // A RGBA color
let {r, g, b} = transparent;  // r == 0.0; g == 0.0; b == 0.0
```

The next example copies global functions of the `Math` object into variables, which might simplify code that does a lot of trigonometry:

```
// Same as const sin=Math.sin, cos=Math.cos, tan=Math.tan
const {sin, cos, tan} = Math;
```

Notice in the code here that the `Math` object has many properties other than the three that are destructured into individual variables. Those that are not named are simply ignored. If the lefthand side of this assignment had included a variable whose name was not a property of `Math`, that variable would simply be assigned `undefined`.

In each of these object destructuring examples, we have chosen variable names that match the property names of the object we're destructuring. This keeps the syntax simple and easy to understand, but it is not required. Each of the identifiers on the lefthand side of an object destructuring assignment can also be a colon-separated pair of identifiers, where the first is the name of the property whose value is to be assigned and the second is the name of the variable to assign it to:

```
// Same as const cosine = Math.cos, tangent = Math.tan;
const { cos: cosine, tan: tangent } = Math;
```

I find that object destructuring syntax becomes too complicated to be useful when the variable names and property names are not the same, and I tend to avoid the shorthand in this case. If you choose to use it, remember that property names are always on the left of the colon, in both object literals and on the left of an object destructuring assignment.

Destructuring assignment becomes even more complicated when it is used with nested objects, or arrays of objects, or objects of arrays, but it is legal:

```
let points = [{x: 1, y: 2}, {x: 3, y: 4}];     // An array of two point objects
let [{x: x1, y: y1}, {x: x2, y: y2}] = points; // destructured into 4 variables.
(x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // => true
```

Or, instead of destructuring an array of objects, we could destructure an object of arrays:

```
let points = { p1: [1,2], p2: [3,4] };         // An object with 2 array props
let { p1: [x1, y1], p2: [x2, y2] } = points;   // destructured into 4 vars
(x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // => true
```

Complex destructuring syntax like this can be hard to write and hard to read, and you may be better off just writing out your assignments explicitly with traditional code like `let x1 = points.p1[0];`.

Understanding Complex Destructuring

If you find yourself working with code that uses complex destructuring assignments, there is a useful regularity that can help you make sense of the complex cases. Think first about a regular (single-value) assignment. After the assignment is done, you can take the variable name from the lefthand side of the assignment and use it as an expression in your code, where it will evaluate to whatever value you assigned it. The same thing is true for destructuring assignment. The lefthand side of a destructuring assignment looks like an array literal or an object literal (§6.2.1 and §6.10). After the assignment has been done, the lefthand side will actually work as a valid array literal or object literal elsewhere in your code. You can check that you've written a destructuring assignment correctly by trying to use the lefthand side on the righthand side of another assignment expression:

```
// Start with a data structure and a complex destructuring
let points = [{x: 1, y: 2}, {x: 3, y: 4}];
let [{x: x1, y: y1}, {x: x2, y: y2}] = points;

// Check your destructuring syntax by flipping the assignment around
let points2 = [{x: x1, y: y1}, {x: x2, y: y2}]; // points2 == points
```

3.11 Summary

Some key points to remember about this chapter:

- How to write and manipulate numbers and strings of text in JavaScript.
- How to work with JavaScript's other primitive types: booleans, Symbols, `null`, and `undefined`.
- The differences between immutable primitive types and mutable reference types.
- How JavaScript converts values implicitly from one type to another and how you can do so explicitly in your programs.
- How to declare and initialize constants and variables (including with destructuring assignment) and the lexical scope of the variables and constants you declare.

[1] This is the format for numbers of type `double` in Java, C++, and most modern programming languages.

[2] There are JavaScript extensions, such as TypeScript and Flow (§17.8), that allow types to be specified as part of variable declarations with syntax like `let x: number = 0;`.