Chapter 5. Statements

Chapter 4 described expressions as JavaScript phrases. By that analogy, *statements* are JavaScript sentences or commands. Just as English sentences are terminated and separated from one another with periods, JavaScript statements are terminated with semicolons (§2.6). Expressions are *evaluated* to produce a value, but statements are *executed* to make something happen.

One way to "make something happen" is to evaluate an expression that has side effects. Expressions with side effects, such as assignments and function invocations, can stand alone as statements, and when used this way are known as *expression statements*. A similar category of statements are the *declaration statements* that declare new variables and define new functions.

JavaScript programs are nothing more than a sequence of statements to execute. By default, the JavaScript interpreter executes these statements one after another in the order they are written. Another way to "make something happen" is to alter this default order of execution, and JavaScript has a number of statements or *control structures* that do just this:

Conditionals

: Statements like `if` and `switch` that make the JavaScript interpreter execute or skip other statements depending on the value of an expression

Loops

: Statements like `while` and `for` that execute other statements repetitively

Jumps

: Statements like `break`, `return`, and `throw` that cause the interpreter to jump to another part of the program

The sections that follow describe the various statements in JavaScript and explain their syntax. Table 5-1, at the end of the chapter, summarizes the syntax. A JavaScript program is simply a sequence of statements, separated from one another with semicolons, so once you are familiar with the statements of JavaScript, you can begin writing JavaScript programs.

5.1 Expression Statements

The simplest kinds of statements in JavaScript are expressions that have side effects. This sort of statement was shown in Chapter 4. Assignment statements are one major category of expression statements. For example:

```
greeting = "Hello " + name;
i *= 3;
```

The increment and decrement operators, `++` and `--`, are related to assignment statements. These have the side effect of changing a variable value, just as if an assignment had been performed:

```
counter++;
```

The `delete` operator has the important side effect of deleting an object property. Thus, it is almost always used as a statement, rather than as part of a larger expression:

```
delete o.x;
```

Function calls are another major category of expression statements. For example:

```
console.log(debugMessage);
displaySpinner(); // A hypothetical function to display a spinner in a web
```

These function calls are expressions, but they have side effects that affect the host environment or program state, and they are used here as statements. If a function does not have any side effects, there is no sense in calling it, unless it is part of a larger expression or an assignment statement. For example, you wouldn't just compute a cosine and discard the result:

```
Math.cos(x);
```

But you might well compute the value and assign it to a variable for future use:

```
cx = Math.cos(x);
```

Note that each line of code in each of these examples is terminated with a semicolon.

5.2 Compound and Empty Statements

Just as the comma operator (§4.13.7) combines multiple expressions into a single expression, a *statement block* combines multiple statements into a single *compound statement*. A statement block is simply a sequence of statements enclosed within curly braces. Thus, the following lines act as a single statement and can be used anywhere that JavaScript expects a single statement:

```
{
    x = Math.PI;
    cx = Math.cos(x);
    console.log("cos(π) = " + cx);
}
```

There are a few things to note about this statement block. First, it does *not* end with a semicolon. The primitive statements within the block end in semicolons, but the block itself does not. Second, the lines inside the block are indented relative to the curly braces that enclose them. This is optional, but it makes the code easier to read and understand.

Just as expressions often contain subexpressions, many JavaScript statements contain substatements. Formally, JavaScript syntax usually allows a single substatement. For example, the `while` loop syntax includes a single statement that serves as the body of the loop. Using a statement block, you can place any number of statements within this single allowed substatement.

A compound statement allows you to use multiple statements where JavaScript syntax expects a single statement. The *empty statement* is the opposite: it allows you to include no statements where one is expected. The empty statement looks like this:

```
;
```

The JavaScript interpreter takes no action when it executes an empty statement. The empty statement is occasionally useful when you want to create a loop that has an empty body. Consider the following `for` loop (`for` loops will be covered in §5.4.3):

```
// Initialize an array a
for(let i = 0; i < a.length; a[i++] = 0) ;
```

In this loop, all the work is done by the expression `a[i++] = 0`, and no loop body is necessary. JavaScript syntax requires a statement as a loop body, however, so an empty statement—just a bare semicolon—is used.

Note that the accidental inclusion of a semicolon after the right parenthesis of a `for` loop, `while` loop, or `if` statement can cause frustrating bugs that are difficult to detect. For example, the following code probably does not do what the author intended:

```
if ((a === 0) || (b === 0));  // Oops! This line does nothing...
    o = null;                 // and this line is always executed.
```

When you intentionally use the empty statement, it is a good idea to comment your code in a way that makes it clear that you are doing it on purpose. For example:

```
for(let i = 0; i < a.length; a[i++] = 0) /* empty */ ;
```

5.3 Conditionals

Conditional statements execute or skip other statements depending on the value of a specified expression. These statements are the decision points of your code, and they are also sometimes known as “branches.” If you imagine a JavaScript interpreter following a path through your code, the conditional statements are the places where the code branches into two or more paths and the interpreter must choose which path to follow.

The following subsections explain JavaScript's basic conditional, the `if/else` statement, and also cover `switch`, a more complicated, multi-way branch statement.

5.3.1 if

The `if` statement is the fundamental control statement that allows JavaScript to make decisions, or, more precisely, to execute statements conditionally. This statement has two forms. The first is:

```
if (expression)
    statement
```

In this form, *expression* is evaluated. If the resulting value is truthy, *statement* is executed. If *expression* is falsy, *statement* is not executed. (See §3.4 for a definition of truthy and falsy values.) For example:

```
if (username == null)       // If username is null or undefined,
    username = "John Doe";  // define it
```

Or similarly:

```
// If username is null, undefined, false, 0, "", or NaN, give it a new valu
if (!username) username = "John Doe";
```

Note that the parentheses around the *expression* are a required part of the syntax for the `if` statement.

JavaScript syntax requires a single statement after the `if` keyword and parenthesized expression, but you can use a statement block to combine multiple statements into one. So the `if` statement might also look like this:

```
if (!address) {
    address = "";
```

```
        message = "Please specify a mailing address.";
    }
```

The second form of the `if` statement introduces an `else` clause that is executed when *expression* is `false`. Its syntax is:

```
if (expression)
    statement1
else
    statement2
```

This form of the statement executes `statement1` if *expression* is truthy and executes `statement2` if *expression* is falsy. For example:

```
if (n === 1)
    console.log("You have 1 new message.");
else
    console.log(`You have ${n} new messages.`);
```

When you have nested `if` statements with `else` clauses, some caution is required to ensure that the `else` clause goes with the appropriate `if` statement. Consider the following lines:

```
i = j = 1;
k = 2;
if (i === j)
    if (j === k)
        console.log("i equals k");
else
    console.log("i doesn't equal j");    // WRONG!!
```

In this example, the inner `if` statement forms the single statement allowed by the syntax of the outer `if` statement. Unfortunately, it is not clear (except from the hint given by the indentation) which `if` the `else` goes with. And in this example, the indentation is wrong, because a JavaScript interpreter actually interprets the previous example as:

```
if (i === j) {
    if (j === k)
        console.log("i equals k");
    else
```

```
            console.log("i doesn't equal j");    // OOPS!
    }
```

The rule in JavaScript (as in most programming languages) is that by default an `else` clause is part of the nearest `if` statement. To make this example less ambiguous and easier to read, understand, maintain, and debug, you should use curly braces:

```
    if (i === j) {
        if (j === k) {
            console.log("i equals k");
        }
    } else {  // What a difference the location of a curly brace makes!
        console.log("i doesn't equal j");
    }
```

Many programmers make a habit of enclosing the bodies of `if` and `else` statements (as well as other compound statements, such as `while` loops) within curly braces, even when the body consists of only a single statement. Doing so consistently can prevent the sort of problem just shown, and I advise you to adopt this practice. In this printed book, I place a premium on keeping example code vertically compact, and I do not always follow my own advice on this matter.

5.3.2 else if

The `if/else` statement evaluates an expression and executes one of two pieces of code, depending on the outcome. But what about when you need to execute one of many pieces of code? One way to do this is with an `else if` statement. `else if` is not really a JavaScript statement, but simply a frequently used programming idiom that results when repeated `if/else` statements are used:

```
    if (n === 1) {
        // Execute code block #1
    } else if (n === 2) {
        // Execute code block #2
    } else if (n === 3) {
        // Execute code block #3
    } else {
        // If all else fails, execute block #4
    }
```

There is nothing special about this code. It is just a series of `if` statements, where each following `if` is part of the `else` clause of the previous statement. Using the `else if` idiom is preferable to, and more legible than, writing these statements out in their syntactically equivalent, fully nested form:

```
if (n === 1) {
    // Execute code block #1
}
else {
    if (n === 2) {
        // Execute code block #2
    }
    else {
        if (n === 3) {
            // Execute code block #3
        }
        else {
            // If all else fails, execute block #4
        }
    }
}
```

5.3.3 switch

An `if` statement causes a branch in the flow of a program's execution, and you can use the `else if` idiom to perform a multiway branch. This is not the best solution, however, when all of the branches depend on the value of the same expression. In this case, it is wasteful to repeatedly evaluate that expression in multiple `if` statements.

The `switch` statement handles exactly this situation. The `switch` keyword is followed by an expression in parentheses and a block of code in curly braces:

```
switch(expression) {
    statements
}
```

However, the full syntax of a `switch` statement is more complex than this. Various locations in the block of code are labeled with the `case` keyword followed by an expression and a colon. When a `switch` executes, it computes the value of *expression* and then looks for a `case` label whose expression evaluates to the same value (where sameness is determined by the `===` operator). If it finds one, it starts executing the block of code at the statement labeled by the `case`. If it does not find a `case` with a matching value, it looks for a statement labeled `default:`. If there is no `default:` label, the `switch` statement skips the block of code altogether.

`switch` is a confusing statement to explain; its operation becomes much clearer with an example. The following `switch` statement is equivalent to the repeated `if/else` statements shown in the previous section:

```
switch(n) {
case 1:                      // Start here if n === 1
    // Execute code block #1.
    break;                   // Stop here
case 2:                      // Start here if n === 2
    // Execute code block #2.
    break;                   // Stop here
case 3:                      // Start here if n === 3
    // Execute code block #3.
    break;                   // Stop here
default:                     // If all else fails...
    // Execute code block #4.
    break;                   // Stop here
}
```

Note the `break` keyword used at the end of each `case` in this code. The `break` statement, described later in this chapter, causes the interpreter to jump to the end (or “break out”) of the `switch` statement and continue with the statement that follows it. The `case` clauses in a `switch` statement specify only the *starting point* of the desired code; they do not specify any ending point. In the absence of `break` statements, a `switch` statement begins executing its block of code at the `case` label that matches the value of its *expression* and continues executing statements until it reaches the end of the block. On rare occasions, it is useful to write code like this that “falls through” from one `case` label to the next, but 99% of the time you should be careful to end every `case` with a `break` statement. (When using `switch` inside a function, however, you may use a `return` statement instead of a `break` statement. Both serve to terminate the `switch` statement and prevent execution from falling through to the next `case`.)

Here is a more realistic example of the `switch` statement; it converts a value to a string in a way that depends on the type of the value:

```
function convert(x) {
    switch(typeof x) {
    case "number":            // Convert the number to a hexadecimal intege
        return x.toString(16);
    case "string":            // Return the string enclosed in quotes
        return '"' + x + '"';
    default:                  // Convert any other type in the usual way
        return String(x);
    }
}
```

Note that in the two previous examples, the `case` keywords are followed by number and string literals, respectively. This is how the `switch` statement is most often used in practice, but note that the ECMAScript standard allows each `case` to be followed by an arbitrary expression.

The `switch` statement first evaluates the expression that follows the `switch` keyword and then evaluates the `case` expressions, in the order in which they appear, until it finds a value that matches.[1] The matching case is determined using the `===` identity operator, not the `==` equality operator, so the expressions must match without any type conversion.

Because not all of the `case` expressions are evaluated each time the `switch` statement is executed, you should avoid using `case` expressions that contain side effects such as function calls or assignments. The safest course is simply to limit your `case` expressions to constant expressions.

As explained earlier, if none of the `case` expressions match the `switch` expression, the `switch` statement begins executing its body at the statement labeled `default:`. If there is no `default:` label, the `switch` statement skips its body altogether. Note that in the examples shown, the `default:` label appears at the end of the `switch` body, following all the `case` labels. This is a logical and common place for it, but it can actually appear anywhere within the body of the statement.

5.4 Loops

To understand conditional statements, we imagined the JavaScript interpreter following a branching path through your source code. The *looping statements* are those that bend that path back upon itself to repeat portions of your code. JavaScript has five looping statements: `while`, `do/while`, `for`, `for/of` (and its `for/await` variant), and `for/in`. The following subsections explain each in turn. One common use for loops is to iterate over the elements of an array. §7.6 discusses this kind of loop in detail and covers special looping methods defined by the Array class.

5.4.1 while

Just as the `if` statement is JavaScript's basic conditional, the `while` statement is JavaScript's basic loop. It has the following syntax:

```
while (expression)
    statement
```

To execute a `while` statement, the interpreter first evaluates *expression*. If the value of the expression is falsy, then the interpreter skips over the *statement* that serves as the loop body and moves on to the next statement in the program. If, on the other hand, the *expression* is truthy, the interpreter executes the *statement* and repeats, jumping back to the top of the loop and evaluating *expression* again. Another way to say this is that the interpreter executes *statement* repeatedly *while* the *expression* is truthy. Note that you can create an infinite loop with the syntax `while(true)`.

Usually, you do not want JavaScript to perform exactly the same operation over and over again. In almost every loop, one or more variables change with each *iteration* of the loop. Since the variables change, the actions performed by executing *statement* may differ each time through the loop. Furthermore, if the changing variable or variables are involved in *expression*, the value of the expression may be different each time through the loop. This is important; otherwise, an expression that starts off truthy would never change, and the loop would never end! Here is an example of a `while` loop that prints the numbers from 0 to 9:

```
let count = 0;
while(count < 10) {
    console.log(count);
    count++;
}
```

As you can see, the variable `count` starts off at 0 and is incremented each time the body of the loop runs. Once the loop has executed 10 times, the expression becomes `false` (i.e., the variable `count` is no longer less than 10), the `while` statement finishes, and the interpreter can move on to the next statement in the program. Many loops have a counter variable like `count`. The variable names `i`, `j`, and `k` are commonly used as loop counters, though you should use more descriptive names if it makes your code easier to understand.

5.4.2 do/while

The `do/while` loop is like a `while` loop, except that the loop expression is tested at the bottom of the loop rather than at the top. This means that the body of the loop is always executed at least once. The syntax is:

```
do
    statement
while (expression);
```

The `do/while` loop is less commonly used than its `while` cousin—in practice, it is somewhat uncommon to be certain that you want a loop to execute at least once. Here's an example of a `do/while` loop:

```
function printArray(a) {
    let len = a.length, i = 0;
    if (len === 0) {
        console.log("Empty Array");
    } else {
        do {
            console.log(a[i]);
        } while(++i < len);
    }
}
```

There are a couple of syntactic differences between the `do/while` loop and the ordinary `while` loop. First, the `do` loop requires both the `do` keyword (to mark the beginning of the loop) and the `while` keyword (to mark the end and introduce the loop condition). Also, the `do` loop must always be terminated with a semicolon. The `while` loop doesn't need a semicolon if the loop body is enclosed in curly braces.

5.4.3 for

The `for` statement provides a looping construct that is often more convenient than the `while` statement. The `for` statement simplifies loops that follow a common pattern. Most loops have a counter variable of some kind. This variable is initialized before the loop starts and is tested before each iteration of the loop. Finally, the counter variable is incremented or otherwise updated at the end of the loop body, just before the variable is tested again. In this kind of loop, the initialization, the test, and the update are the three crucial manipulations of a loop variable. The `for` statement encodes each of these three manipulations as an expression and makes those expressions an explicit part of the loop syntax:

```
for(initialize ; test ; increment)
    statement
```

initialize, *test*, and *increment* are three expressions (separated by semicolons) that are responsible for initializing, testing, and incrementing the loop variable. Putting them all in the first line of the loop makes it easy to understand what a `for` loop is doing and prevents mistakes such as forgetting to initialize or increment the loop variable.

The simplest way to explain how a `for` loop works is to show the equivalent `while` loop:[2]

```
initialize;
while(test) {
    statement
    increment;
}
```

In other words, the *initialize* expression is evaluated once, before the loop begins. To be useful, this expression must have side effects (usually an assignment). JavaScript also allows *initialize* to be a variable declaration statement so that you can declare and initialize a loop counter at the same time. The *test* expression is evaluated before each iteration and controls whether the body of the loop is executed. If *test* evaluates to a truthy value, the *statement* that is the body of the loop is executed. Finally, the *increment* expression is evaluated. Again, this must be an expression with side effects in order to be useful. Generally, it is either an assignment expression, or it uses the `++` or `--` operators.

We can print the numbers from 0 to 9 with a `for` loop like the following. Contrast it with the equivalent `while` loop shown in the previous section:

```
for(let count = 0; count < 10; count++) {
    console.log(count);
}
```

Loops can become a lot more complex than this simple example, of course, and sometimes multiple variables change with each iteration of the loop. This situation is the only place that the comma operator is commonly used in JavaScript; it provides a way to combine multiple initialization and increment expressions into a single expression suitable for use in a `for` loop:

```
let i, j, sum = 0;
for(i = 0, j = 10 ; i < 10 ; i++, j--) {
    sum += i * j;
}
```

In all our loop examples so far, the loop variable has been numeric. This is quite common but is not necessary. The following code uses a `for` loop to traverse a linked list data structure and return the last object in the list (i.e., the first object that does not have a `next` property):

```
function tail(o) {                          // Return the tail of linked li
    for(; o.next; o = o.next) /* empty */ ; // Traverse while o.next is tru
    return o;
}
```

Note that this code has no *initialize* expression. Any of the three expressions may be omitted from a `for` loop, but the two semicolons are required. If you omit the *test* expression, the loop repeats forever, and `for(;;)` is another way of writing an infinite loop, like `while(true)`.

5.4.4 for/of

ES6 defines a new loop statement: `for/of`. This new kind of loop uses the `for` keyword but is a completely different kind of loop than the regular `for` loop. (It is also completely different than the older `for/in` loop that we'll describe in §5.4.5.)

The `for/of` loop works with *iterable* objects. We'll explain exactly what it means for an object to be iterable in Chapter 12, but for this chapter, it is enough to know that arrays, strings, sets, and maps are iterable: they represent a sequence or set of elements that you can loop or iterate through using a `for/of` loop.

Here, for example, is how we can use `for/of` to loop through the elements of an array of numbers and compute their sum:

```
let data = [1, 2, 3, 4, 5, 6, 7, 8, 9], sum = 0;
for(let element of data) {
    sum += element;
}
sum       // => 45
```

Superficially, the syntax looks like a regular `for` loop: the `for` keyword is followed by parentheses that contain details about what the loop should do. In this case, the parentheses contain a variable declaration (or, for variables that have already been declared, simply the name of the variable) followed by the `of` keyword and an expression that evaluates to an iterable object, like the `data` array in this case. As with all loops, the body of a `for/of` loop follows the parentheses, typically within curly braces.

In the code just shown, the loop body runs once for each element of the `data` array. Before each execution of the loop body, the next element of the array is assigned to the element variable. Array elements are iterated in order from first to last.

Arrays are iterated "live"—changes made during the iteration may affect the outcome of the iteration. If we modify the preceding code by adding the line `data.push(sum);` inside the loop body, then we create an infinite loop because the iteration can never reach the last element of the array.

for/of with objects

Objects are not (by default) iterable. Attempting to use `for/of` on a regular object throws a TypeError at runtime:

```
let o = { x: 1, y: 2, z: 3 };
for(let element of o) { // Throws TypeError because o is not iterable
    console.log(element);
}
```

If you want to iterate through the properties of an object, you can use the `for/in` loop (introduced in §5.4.5), or use `for/of` with the `Object.keys()` method:

```
let o = { x: 1, y: 2, z: 3 };
let keys = "";
for(let k of Object.keys(o)) {
    keys += k;
}
keys  // => "xyz"
```

This works because `Object.keys()` returns an array of property names for an object, and arrays are iterable with `for/of`. Note also that this iteration of the keys of an object is not live as the array example above was —changes to the object `o` made in the loop body will have no effect on the iteration. If you don't care about the keys of an object, you can also iterate through their corresponding values like this:

```
let sum = 0;
for(let v of Object.values(o)) {
    sum += v;
}
sum // => 6
```

And if you are interested in both the keys and the values of an object's properties, you can use `for/of` with `Object.entries()` and destructuring assignment:

```
let pairs = "";
for(let [k, v] of Object.entries(o)) {
    pairs += k + v;
}
pairs  // => "x1y2z3"
```

`Object.entries()` returns an array of arrays, where each inner array represents a key/value pair for one property of the object. We use destructuring assignment in this code example to unpack those inner arrays into two individual variables.

for/of with strings

Strings are iterable character-by-character in ES6:

```
let frequency = {};
for(let letter of "mississippi") {
    if (frequency[letter]) {
        frequency[letter]++;
    } else {
        frequency[letter] = 1;
    }
}
frequency   // => {m: 1, i: 4, s: 4, p: 2}
```

Note that strings are iterated by Unicode codepoint, not by UTF-16 character. The string "I ❤ 🐈" has a `.length` of 5 (because the two emoji characters each require two UTF-16 characters to represent). But if you iterate that string with `for/of`, the loop body will run three times, once for each of the three code points "I", "❤", and "🐈."

for/of with Set and Map

The built-in ES6 Set and Map classes are iterable. When you iterate a Set with `for/of`, the loop body runs once for each element of the set. You could use code like this to print the unique words in a string of text:

```
let text = "Na na na na na na na na Batman!";
let wordSet = new Set(text.split(" "));
let unique = [];
for(let word of wordSet) {
    unique.push(word);
}
unique // => ["Na", "na", "Batman!"]
```

Maps are an interesting case because the iterator for a Map object does not iterate the Map keys, or the Map values, but key/value pairs. Each time through the iteration, the iterator returns an array whose first element is a key and whose second element is the corresponding value. Given a Map `m`, you could iterate and destructure its key/value pairs like this:

```
let m = new Map([[1, "one"]]);
for(let [key, value] of m) {
    key     // => 1
    value   // => "one"
}
```

Asynchronous iteration with for/await

ES2018 introduces a new kind of iterator, known as an *asynchronous iterator*, and a variant on the `for/of` loop, known as the `for/await` loop that works with asynchronous iterators.

You'll need to read Chapters 12 and 13 in order to understand the `for/await` loop, but here is how it looks in code:

```
// Read chunks from an asynchronously iterable stream and print them out
async function printStream(stream) {
    for await (let chunk of stream) {
        console.log(chunk);
    }
}
```

5.4.5 for/in

A `for/in` loop looks a lot like a `for/of` loop, with the `of` keyword changed to `in`. While a `for/of` loop requires an iterable object after the `of`, a `for/in` loop works with any object after the `in`. The `for/of` loop is new in ES6, but `for/in` has been part of JavaScript since the very beginning (which is why it has the more natural sounding syntax).

The `for/in` statement loops through the property names of a specified object. The syntax looks like this:

```
for (variable in object)
    statement
```

variable typically names a variable, but it may be a variable declaration or anything suitable as the left-hand side of an assignment expression. *object* is an expression that evaluates to an object. As usual, *statement* is the statement or statement block that serves as the body of the loop.

And you might use a `for/in` loop like this:

```
for(let p in o) {       // Assign property names of o to variable p
    console.log(o[p]); // Print the value of each property
}
```

To execute a `for/in` statement, the JavaScript interpreter first evaluates the *object* expression. If it evaluates to `null` or `undefined`, the interpreter skips the loop and moves on to the next statement. The interpreter now executes the body of the loop once for each enumerable property of the object. Before each iteration, however, the interpreter evaluates the *variable* expression and assigns the name of the property (a string value) to it.

Note that the *variable* in the `for/in` loop may be an arbitrary expression, as long as it evaluates to something suitable for the left side of an assignment. This expression is evaluated each time through the loop, which means that it may evaluate differently each time. For example, you can use code like the following to copy the names of all object properties into an array:

```
let o = { x: 1, y: 2, z: 3 };
let a = [], i = 0;
for(a[i++] in o) /* empty */;
```

JavaScript arrays are simply a specialized kind of object, and array indexes are object properties that can be enumerated with a `for/in` loop. For example, following the previous code with this line enumerates the array indexes 0, 1, and 2:

```
for(let i in a) console.log(i);
```

I find that a common source of bugs in my own code is the accidental use of `for/in` with arrays when I meant to use `for/of`. When working with arrays, you almost always want to use `for/of` instead of `for/in`.

The `for/in` loop does not actually enumerate all properties of an object. It does not enumerate properties whose names are symbols. And of the properties whose names are strings, it only loops over the *enumerable* properties (see §14.1). The various built-in methods defined by core JavaScript are not enumerable. All objects have a `toString()` method, for example, but the `for/in` loop does not enumerate this `toString` property. In addition to built-in methods, many other properties of the built-in objects are non-enumerable. All properties and methods defined by your code are enumerable, by default. (You can make them non-enumerable using techniques explained in §14.1.)

Enumerable inherited properties (see §6.3.2) are also enumerated by the `for/in` loop. This means that if you use `for/in` loops and also use code that defines properties that are inherited by all objects, then your loop may not behave in the way you expect. For this reason, many programmers prefer to use a `for/of` loop with `Object.keys()` instead of a `for/in` loop.

If the body of a `for/in` loop deletes a property that has not yet been enumerated, that property will not be enumerated. If the body of the loop defines new properties on the object, those properties may or may not be enumerated. See §6.6.1 for more information on the order in which `for/in` enumerates the properties of an object.

5.5 Jumps

Another category of JavaScript statements are *jump statements*. As the name implies, these cause the JavaScript interpreter to jump to a new location in the source code. The `break` statement makes the interpreter jump to the end of a loop or other statement. `continue` makes the interpreter skip the rest of the body of a loop and jump back to the top of a loop to begin a new iteration. JavaScript allows statements to be named, or *labeled*, and `break` and `continue` can identify the target loop or other statement label.

The `return` statement makes the interpreter jump from a function invocation back to the code that invoked it and also supplies the value for the invocation. The `throw` statement is a kind of interim return from a generator function. The `throw` statement raises, or *throws*, an exception and is designed to work with the `try/catch/finally` statement, which establishes a block of exception-handling code. This is a complicated kind of jump statement: when an exception is thrown, the interpreter jumps to the nearest enclosing exception handler, which may be in the same function or up the call stack in an invoking function.

Details about each of these jump statements are in the sections that follow.

5.5.1 Labeled Statements

Any statement may be *labeled* by preceding it with an identifier and a colon:

```
identifier: statement
```

By labeling a statement, you give it a name that you can use to refer to it elsewhere in your program. You can label any statement, although it is only useful to label statements that have bodies, such as loops and conditionals. By giving a loop a name, you can use `break` and `continue` statements inside the body of the loop to exit the loop or to jump directly to the top of the loop to begin the next iteration. `break` and `continue` are the only JavaScript statements that use statement labels; they are covered in the following subsections. Here is an example of a labeled `while` loop and a `continue` statement that uses the label.

```
mainloop: while(token !== null) {
    // Code omitted...
    continue mainloop;  // Jump to the next iteration of the named loop
    // More code omitted...
}
```

The *identifier* you use to label a statement can be any legal JavaScript identifier that is not a reserved word. The namespace for labels is different than the namespace for variables and functions, so you can use the same identifier as a statement label and as a variable or function name. Statement labels are defined only within the statement to which they apply (and within its substatements, of course). A statement may not have the same label as a statement that contains it, but two statements may have the same label as long as neither one is nested within the other. Labeled statements may themselves be labeled. Effectively, this means that any statement may have multiple labels.

5.5.2 break

The `break` statement, used alone, causes the innermost enclosing loop or `switch` statement to exit immediately. Its syntax is simple:

```
break;
```

Because it causes a loop or `switch` to exit, this form of the `break` statement is legal only if it appears inside one of these statements.

You've already seen examples of the `break` statement within a `switch` statement. In loops, it is typically used to exit prematurely when, for whatever reason, there is no longer any need to complete the loop. When a loop has complex termination conditions, it is often easier to implement some of these conditions with `break` statements rather than trying to express them all in a single loop expression. The following code searches the elements of an array for a particular value. The loop terminates in the normal way when it reaches the end of the array; it terminates with a `break` statement if it finds what it is looking for in the array:

```
for(let i = 0; i < a.length; i++) {
    if (a[i] === target) break;
}
```

JavaScript also allows the `break` keyword to be followed by a statement label (just the identifier, with no colon):

```
break labelname;
```

When `break` is used with a label, it jumps to the end of, or terminates, the enclosing statement that has the specified label. It is a syntax error to use `break` in this form if there is no enclosing statement with the specified label. With this form of the `break` statement, the named statement need not be a loop or `switch`: `break` can "break out of" any enclosing statement. This statement can even be a statement block grouped within curly braces for the sole purpose of naming the block with a label.

A newline is not allowed between the `break` keyword and the *labelname*. This is a result of JavaScript's automatic insertion of omitted semicolons: if you put a line terminator between the `break` keyword and the label that follows, JavaScript assumes you meant to use the simple, unlabeled form of the statement and treats the line terminator as a semicolon. (See §2.6.)

You need the labeled form of the `break` statement when you want to break out of a statement that is not the nearest enclosing loop or a `switch`. The following code demonstrates:

```
let matrix = getData();  // Get a 2D array of numbers from somewhere
// Now sum all the numbers in the matrix.
let sum = 0, success = false;
// Start with a labeled statement that we can break out of if errors occur
computeSum: if (matrix) {
    for(let x = 0; x < matrix.length; x++) {
        let row = matrix[x];
        if (!row) break computeSum;
        for(let y = 0; y < row.length; y++) {
            let cell = row[y];
            if (isNaN(cell)) break computeSum;
            sum += cell;
        }
    }
    success = true;
}
// The break statements jump here. If we arrive here with success == false
// then there was something wrong with the matrix we were given.
// Otherwise, sum contains the sum of all cells of the matrix.
```

Finally, note that a `break` statement, with or without a label, can not transfer control across function boundaries. You cannot label a function definition statement, for example, and then use that label inside the function.

5.5.3 continue

The `continue` statement is similar to the `break` statement. Instead of exiting a loop, however, `continue` restarts a loop at the next iteration. The `continue` statement's syntax is just as simple as the `break` statement's:

```
continue;
```

The `continue` statement can also be used with a label:

```
continue labelname;
```

The `continue` statement, in both its labeled and unlabeled forms, can be used only within the body of a loop. Using it anywhere else causes a syntax error.

When the `continue` statement is executed, the current iteration of the enclosing loop is terminated, and the next iteration begins. This means different things for different types of loops:

- In a `while` loop, the specified *expression* at the beginning of the loop is tested again, and if it's `true`, the loop body is executed starting from the top.
- In a `do/while` loop, execution skips to the bottom of the loop, where the loop condition is tested again before restarting the loop at the top.
- In a `for` loop, the *increment* expression is evaluated, and the *test* expression is tested again to determine if another iteration should be done.
- In a `for/of` or `for/in` loop, the loop starts over with the next iterated value or next property name being assigned to the specified variable.

Note the difference in behavior of the `continue` statement in the `while` and `for` loops: a `while` loop returns directly to its condition, but a `for` loop first evaluates its *increment* expression and then returns to its condition. Earlier, we considered the behavior of the `for` loop in terms of an “equivalent” `while` loop. Because the `continue` statement behaves differently for these two loops, however, it is not actually possible to perfectly simulate a `for` loop with a `while` loop alone.

The following example shows an unlabeled `continue` statement being used to skip the rest of the current iteration of a loop when an error occurs:

```
for(let i = 0; i < data.length; i++) {
    if (!data[i]) continue;  // Can't proceed with undefined data
    total += data[i];
}
```

Like the `break` statement, the `continue` statement can be used in its labeled form within nested loops when the loop to be restarted is not the immediately enclosing loop. Also, as with the `break` statement, line breaks are not allowed between the `continue` statement and its *labelname*.

5.5.4 return

Recall that function invocations are expressions and that all expressions have values. A `return` statement within a function specifies the value of invocations of that function. Here’s the syntax of the `return` statement:

```
return expression;
```

A `return` statement may appear only within the body of a function. It is a syntax error for it to appear anywhere else. When the `return` statement is executed, the function that contains it returns the value of *expression* to its caller. For example:

```
function square(x) { return x*x; } // A function that has a return statemen
square(2)                          // => 4
```

With no `return` statement, a function invocation simply executes each of the statements in the function body in turn until it reaches the end of the function and then returns to its caller. In this case, the invocation expression evaluates to `undefined`. The `return` statement often appears as the last statement in a function, but it need not be last: a function returns to its caller when a `return` statement is executed, even if there are other statements remaining in the function body.

The `return` statement can also be used without an *expression* to make the function return `undefined` to its caller. For example:

```
function displayObject(o) {
    // Return immediately if the argument is null or undefined.
    if (!o) return;
    // Rest of function goes here...
}
```

Because of JavaScript's automatic semicolon insertion (§2.6), you cannot include a line break between the `return` keyword and the expression that follows it.

5.5.5 yield

The `yield` statement is much like the `return` statement but is used only in ES6 generator functions (see §12.3) to produce the next value in the generated sequence of values without actually returning:

```
// A generator function that yields a range of integers
function* range(from, to) {
    for(let i = from; i <= to; i++) {
        yield i;
    }
}
```

In order to understand `yield`, you must understand iterators and generators, which will not be covered until Chapter 12. `yield` is included here for completeness, however. (Technically, though, `yield` is an operator rather than a statement, as explained in §12.4.2.)

5.5.6 throw

An *exception* is a signal that indicates that some sort of exceptional condition or error has occurred. To *throw* an exception is to signal such an error or exceptional condition. To *catch* an exception is to handle it—to take whatever actions are necessary or appropriate to recover from the exception. In JavaScript, exceptions are thrown whenever a runtime error occurs and whenever the program explicitly throws one using the `throw` statement. Exceptions are caught with the `try/catch/finally` statement, which is described in the next section.

The `throw` statement has the following syntax:

```
throw expression;
```

expression may evaluate to a value of any type. You might throw a number that represents an error code or a string that contains a human-readable error message. The Error class and its subclasses are used when the JavaScript interpreter itself throws an error, and you can use them as well. An Error object has a `name` property that specifies the type of error and a `message` property that holds the string passed to the constructor function. Here is an example function that throws an Error object when invoked with an invalid argument:

```
function factorial(x) {
    // If the input argument is invalid, throw an exception!
    if (x < 0) throw new Error("x must not be negative");
    // Otherwise, compute a value and return normally
    let f;
    for(f = 1; x > 1; f *= x, x--) /* empty */ ;
    return f;
}
factorial(4)   // => 24
```

When an exception is thrown, the JavaScript interpreter immediately stops normal program execution and jumps to the nearest exception handler. Exception handlers are written using the `catch` clause of the `try/catch/finally` statement, which is described in the next section. If the block of code in which the exception was thrown does not have an associated `catch` clause, the interpreter checks the next-highest enclosing block of code to see if it has an exception handler associated with it. This continues until a handler is found. If an exception is thrown in a function that does not contain a `try/catch/finally` statement to handle it, the exception propagates up to the code that invoked the function. In this way, exceptions propagate up through the lexical structure of JavaScript methods and up the call stack. If no exception handler is ever found, the exception is treated as an error and is reported to the user.

5.5.7 try/catch/finally

The `try/catch/finally` statement is JavaScript's exception handling mechanism. The `try` clause of this statement simply defines the block of code whose exceptions are to be handled. The `try` block is followed by a `catch` clause, which is a block of statements that are invoked when an exception occurs anywhere within the `try` block. The `catch` clause is followed by a `finally` block containing cleanup code that is guaranteed to be executed, regardless of what happens in the `try` block. Both the `catch` and `finally` blocks are optional, but a `try` block must be accompanied by at least one of these blocks. The `try`, `catch`, and `finally` blocks all begin and end with curly braces. These braces are a required part of the syntax and cannot be omitted, even if a clause contains only a single statement.

The following code illustrates the syntax and purpose of the `try/catch/finally` statement:

```
try {
    // Normally, this code runs from the top of the block to the bottom
    // without problems. But it can sometimes throw an exception,
    // either directly, with a throw statement, or indirectly, by calling
    // a method that throws an exception.
}
catch(e) {
    // The statements in this block are executed if, and only if, the try
    // block throws an exception. These statements can use the local variab
    // e to refer to the Error object or other value that was thrown.
    // This block may handle the exception somehow, may ignore the
    // exception by doing nothing, or may rethrow the exception with throw.
}
finally {
    // This block contains statements that are always executed, regardless
    // what happens in the try block. They are executed whether the try
    // block terminates:
    //   1) normally, after reaching the bottom of the block
    //   2) because of a break, continue, or return statement
    //   3) with an exception that is handled by a catch clause above
    //   4) with an uncaught exception that is still propagating
}
```

Note that the `catch` keyword is generally followed by an identifier in parentheses. This identifier is like a function parameter. When an exception is caught, the value associated with the exception (an Error object, for example) is assigned to this parameter. The identifier associated with a `catch` clause has block scope—it is only defined within the `catch` block.

Here is a realistic example of the `try/catch` statement. It uses the `factorial()` method defined in the previous section and the client-side JavaScript methods `prompt()` and `alert()` for input and output:

```
try {
    // Ask the user to enter a number
    let n = Number(prompt("Please enter a positive integer", ""));
    // Compute the factorial of the number, assuming the input is valid
    let f = factorial(n);
    // Display the result
    alert(n + "! = " + f);
}
catch(ex) {     // If the user's input was not valid, we end up here
    alert(ex);  // Tell the user what the error is
}
```

This example is a `try/catch` statement with no `finally` clause. Although `finally` is not used as often as `catch`, it can be useful. However, its behavior requires additional explanation. The `finally` clause is guaranteed to be executed if any portion of the `try` block is executed, regardless of how the code in the `try` block completes. It is generally used to clean up after the code in the `try` clause.

In the normal case, the JavaScript interpreter reaches the end of the `try` block and then proceeds to the `finally` block, which performs any necessary cleanup. If the interpreter left the `try` block because of a `return`, `continue`, or `break` statement, the `finally` block is executed before the interpreter jumps to its new destination.

If an exception occurs in the `try` block and there is an associated `catch` block to handle the exception, the interpreter first executes the `catch` block and then the `finally` block. If there is no local `catch` block to handle the exception, the interpreter first executes the `finally` block and then jumps to the nearest containing `catch` clause.

If a `finally` block itself causes a jump with a `return`, `continue`, `break`, or `throw` statement, or by calling a method that throws an exception, the interpreter abandons whatever jump was pending and performs the new jump. For example, if a `finally` clause throws an exception, that exception replaces any exception that was in the process of being thrown. If a `finally` clause issues a `return` statement, the method returns normally, even if an exception has been thrown and has not yet been handled.

`try` and `finally` can be used together without a `catch` clause. In this case, the `finally` block is simply cleanup code that is guaranteed to be executed, regardless of what happens in the `try` block. Recall that we can't completely simulate a `for` loop with a `while` loop because the `continue` statement behaves differently for the two loops. If we add a `try/finally` statement, we can write a `while` loop that works like a `for` loop and that handles `continue` statements correctly:

```
// Simulate for(initialize ; test ;increment ) body;
initialize ;
while( test ) {
    try { body ; }
    finally { increment ; }
}
```

Note, however, that a *body* that contains a `break` statement behaves slightly differently (causing an extra increment before exiting) in the `while` loop than it does in the `for` loop, so even with the `finally` clause, it is not possible to completely simulate the `for` loop with `while`.

BARE CATCH CLAUSES

Occasionally you may find yourself using a `catch` clause solely to detect and stop the propagation of an exception, even though you do not care about the type or the value of the exception. In ES2019 and later, you can omit the parentheses and the identifier and use the `catch` keyword bare in this case. Here is an example:

```
// Like JSON.parse(), but return undefined instead of throwing an error
function parseJSON(s) {
    try {
        return JSON.parse(s);
    } catch {
        // Something went wrong but we don't care what it was
        return undefined;
    }
}
```

5.6 Miscellaneous Statements

This section describes the remaining three JavaScript statements— `with`, `debugger`, and `"use strict"`.

5.6.1 with

The `with` statement runs a block of code as if the properties of a specified object were variables in scope for that code. It has the following syntax:

```
with (object)
    statement
```

This statement creates a temporary scope with the properties of *object* as variables and then executes *statement* within that scope.

The `with` statement is forbidden in strict mode (see §5.6.3) and should be considered deprecated in non-strict mode: avoid using it whenever possible. JavaScript code that uses `with` is difficult to optimize and is likely to run significantly more slowly than the equivalent code written without the `with` statement.

The common use of the `with` statement is to make it easier to work with deeply nested object hierarchies. In client-side JavaScript, for example, you may have to type expressions like this one to access elements of an HTML form:

```
document.forms[0].address.value
```

If you need to write expressions like this a number of times, you can use the `with` statement to treat the properties of the form object like variables:

```
with(document.forms[0]) {
    // Access form elements directly here. For example:
    name.value = "";
    address.value = "";
    email.value = "";
}
```

This reduces the amount of typing you have to do: you no longer need to prefix each form property name with `document.forms[0]`. It is just as simple, of course, to avoid the `with` statement and write the preceding code like this:

```
let f = document.forms[0];
f.name.value = "";
f.address.value = "";
f.email.value = "";
```

Note that if you use `const` or `let` or `var` to declare a variable or constant within the body of a `with` statement, it creates an ordinary variable and does not define a new property within the specified object.

5.6.2 debugger

The `debugger` statement normally does nothing. If, however, a debugger program is available and is running, then an implementation may (but is not required to) perform some kind of debugging action. In practice, this statement acts like a breakpoint: execution of JavaScript code stops, and you can use the debugger to print variables' values, examine the call stack, and so on. Suppose, for example, that you are getting an exception in your function `f()` because it is being called with an undefined argument, and you can't figure out where this call is coming from. To help you in debugging this problem, you might alter `f()` so that it begins like this:

```
function f(o) {
  if (o === undefined) debugger;  // Temporary line for debugging purposes
  ...                             // The rest of the function goes here.
}
```

Now, when `f()` is called with no argument, execution will stop, and you can use the debugger to inspect the call stack and find out where this incorrect call is coming from.

Note that it is not enough to have a debugger available: the `debugger` statement won't start the debugger for you. If you're using a web browser and have the developer tools console open, however, this statement will cause a breakpoint.

5.6.3 "use strict"

`"use strict"` is a *directive* introduced in ES5. Directives are not statements (but are close enough that `"use strict"` is documented here). There are two important differences between the `"use strict"` directive and regular statements:

- It does not include any language keywords: the directive is just an expression statement that consists of a special string literal (in single or double quotes).
- It can appear only at the start of a script or at the start of a function body, before any real statements have appeared.

The purpose of a `"use strict"` directive is to indicate that the code that follows (in the script or function) is *strict code*. The top-level (non-function) code of a script is strict code if the script has a `"use strict"` directive. A function body is strict code if it is defined within strict code or if it has a `"use strict"` directive. Code passed to the `eval()` method is strict code if `eval()` is called from strict code or if the string of code includes a `"use strict"` directive. In addition to code explicitly declared to be strict, any code in a `class` body (Chapter 9) or in an ES6 module (§10.3) is automatically strict code. This means that if all of your

JavaScript code is written as modules, then it is all automatically strict, and you will never need to use an explicit `"use strict"` directive.

Strict code is executed in *strict mode*. Strict mode is a restricted subset of the language that fixes important language deficiencies and provides stronger error checking and increased security. Because strict mode is not the default, old JavaScript code that still uses the deficient legacy features of the language will continue to run correctly. The differences between strict mode and non-strict mode are the following (the first three are particularly important):

- The `with` statement is not allowed in strict mode.
- In strict mode, all variables must be declared: a ReferenceError is thrown if you assign a value to an identifier that is not a declared variable, function, function parameter, `catch` clause parameter, or property of the global object. (In non-strict mode, this implicitly declares a global variable by adding a new property to the global object.)
- In strict mode, functions invoked as functions (rather than as methods) have a `this` value of `undefined`. (In non-strict mode, functions invoked as functions are always passed the global object as their `this` value.) Also, in strict mode, when a function is invoked with `call()` or `apply()` (§8.7.4), the `this` value is exactly the value passed as the first argument to `call()` or `apply()`. (In non-strict mode, `null` and `undefined` values are replaced with the global object and nonobject values are converted to objects.)
- In strict mode, assignments to nonwritable properties and attempts to create new properties on non-extensible objects throw a TypeError. (In non-strict mode, these attempts fail silently.)
- In strict mode, code passed to `eval()` cannot declare variables or define functions in the caller's scope as it can in non-strict mode. Instead, variable and function definitions live in a new scope created for the `eval()`. This scope is discarded when the `eval()` returns.
- In strict mode, the Arguments object (§8.3.3) in a function holds a static copy of the values passed to the function. In non-strict mode, the Arguments object has "magical" behavior in which elements of the array and named function parameters both refer to the same value.
- In strict mode, a SyntaxError is thrown if the `delete` operator is followed by an unqualified identifier such as a variable, function, or function parameter. (In nonstrict mode, such a `delete` expression does nothing and evaluates to `false`.)

- In strict mode, an attempt to delete a nonconfigurable property throws a TypeError. (In non-strict mode, the attempt fails and the `delete` expression evaluates to `false`.)
- In strict mode, it is a syntax error for an object literal to define two or more properties by the same name. (In non-strict mode, no error occurs.)
- In strict mode, it is a syntax error for a function declaration to have two or more parameters with the same name. (In non-strict mode, no error occurs.)
- In strict mode, octal integer literals (beginning with a 0 that is not followed by an x) are not allowed. (In non-strict mode, some implementations allow octal literals.)
- In strict mode, the identifiers `eval` and `arguments` are treated like keywords, and you are not allowed to change their value. You cannot assign a value to these identifiers, declare them as variables, use them as function names, use them as function parameter names, or use them as the identifier of a `catch` block.
- In strict mode, the ability to examine the call stack is restricted. `arguments.caller` and `arguments.callee` both throw a TypeError within a strict mode function. Strict mode functions also have `caller` and `arguments` properties that throw TypeError when read. (Some implementations define these nonstandard properties on non-strict functions.)

5.7 Declarations

The keywords `const`, `let`, `var`, `function`, `class`, `import`, and `export` are not technically statements, but they look a lot like statements, and this book refers informally to them as statements, so they deserve a mention in this chapter.

These keywords are more accurately described as *declarations* rather than statements. We said at the start of this chapter that statements “make something happen.” Declarations serve to define new values and give them names that we can use to refer to those values. They don’t make much happen themselves, but by providing names for values they, in an important sense, define the meaning of the other statements in your program.

When a program runs, it is the program's expressions that are being evaluated and the program's statements that are being executed. The declarations in a program don't "run" in the same way: instead, they define the structure of the program itself. Loosely, you can think of declarations as the parts of the program that are processed before the code starts running.

JavaScript declarations are used to define constants, variables, functions, and classes and for importing and exporting values between modules. The next subsections give examples of all of these declarations. They are all covered in much more detail elsewhere in this book.

5.7.1 const, let, and var

The `const`, `let`, and `var` declarations are covered in §3.10. In ES6 and later, `const` declares constants, and `let` declares variables. Prior to ES6, the `var` keyword was the only way to declare variables and there was no way to declare constants. Variables declared with `var` are scoped to the containing function rather than the containing block. This can be a source of bugs, and in modern JavaScript there is really no reason to use `var` instead of `let`.

```
const TAU = 2*Math.PI;
let radius = 3;
var circumference = TAU * radius;
```

5.7.2 function

The `function` declaration is used to define functions, which are covered in detail in Chapter 8. (We also saw `function` in §4.3, where it was used as part of a function expression rather than a function declaration.) A function declaration looks like this:

```
function area(radius) {
    return Math.PI * radius * radius;
}
```

A function declaration creates a function object and assigns it to the specified name— `area` in this example. Elsewhere in our program, we can refer to the function—and run the code inside it—by using this name. The function declarations in any block of JavaScript code are processed before that code runs, and the function names are bound to the function objects throughout the block. We say that function declarations are "hoisted" because it is as if they had all been moved up to the top of whatever scope they are defined within. The upshot is that code that invokes a function can exist in your program before the code that declares the function.

§12.3 describes a special kind of function known as a *generator*. Generator declarations use the `function` keyword but follow it with an asterisk. §13.3 describes asynchronous functions, which are also declared using the `function` keyword but are prefixed with the `async` keyword.

5.7.3 class

In ES6 and later, the `class` declaration creates a new class and gives it a name that we can use to refer to it. Classes are described in detail in Chapter 9. A simple class declaration might look like this:

```
class Circle {
    constructor(radius) { this.r = radius; }
    area() { return Math.PI * this.r * this.r; }
    circumference() { return 2 * Math.PI * this.r; }
}
```

Unlike functions, class declarations are not hoisted, and you cannot use a class declared this way in code that appears before the declaration.

5.7.4 import and export

The `import` and `export` declarations are used together to make values defined in one module of JavaScript code available in another module. A module is a file of JavaScript code with its own global namespace, completely independent of all other modules. The only way that a value (such as function or class) defined in one module can be used in another module is if the defining module exports it with `export` and the using module imports it with `import`. Modules are the subject of Chapter 10, and `import` and `export` are covered in detail in §10.3.

`import` directives are used to import one or more values from another file of JavaScript code and give them names within the current module.

`import` directives come in a few different forms. Here are some examples:

```
import Circle from './geometry/circle.js';
import { PI, TAU } from './geometry/constants.js';
import { magnitude as hypotenuse } from './vectors/utils.js';
```

Values within a JavaScript module are private and cannot be imported into other modules unless they have been explicitly exported. The `export` directive does this: it declares that one or more values defined in the current module are exported and therefore available for import by other modules. The `export` directive has more variants than the `import` directive does. Here is one of them:

```
// geometry/constants.js
const PI = Math.PI;
const TAU = 2 * PI;
export { PI, TAU };
```

The `export` keyword is sometimes used as a modifier on other declarations, resulting in a kind of compound declaration that defines a constant, variable, function, or class and exports it at the same time. And when a module exports only a single value, this is typically done with the special form `export default`:

```
export const TAU = 2 * Math.PI;
export function magnitude(x,y) { return Math.sqrt(x*x + y*y); }
export default class Circle { /* class definition omitted here */ }
```

5.8 Summary of JavaScript Statements

This chapter introduced each of the JavaScript language's statements, which are summarized in Table 5-1.

Table 5-1. JavaScript statement syntax

Statement	Purpose
break	Exit from the innermost loop or `switch` or from named enclosing statement
case	Label a statement within a `switch`
class	Declare a class
const	Declare and initialize one or more constants
continue	Begin next iteration of the innermost loop or the named loop
debugger	Debugger breakpoint
default	Label the default statement within a `switch`
do/while	An alternative to the `while` loop
export	Declare values that can be imported into other modules
for	An easy-to-use loop
for/await	Asynchronously iterate the values of an async iterator
for/in	Enumerate the property names of an object
for/of	Enumerate the values of an iterable object such as an array
function	Declare a function
if/else	Execute one statement or another depending on a condition

Statement	Purpose
import	Declare names for values defined in other modules
label	Give statement a name for use with `break` and `continue`
let	Declare and initialize one or more block-scoped variables (new syntax)
return	Return a value from a function
switch	Multiway branch to `case` or `default:` labels
throw	Throw an exception
try/catch/finally	Handle exceptions and code cleanup
“use strict”	Apply strict mode restrictions to script or function
var	Declare and initialize one or more variables (old syntax)
while	A basic loop construct
with	Extend the scope chain (deprecated and forbidden in strict mode)
yield	Provide a value to be iterated; only used in generator functions

[1] The fact that the `case` expressions are evaluated at runtime makes the JavaScript `switch` statement much different from (and less efficient than) the `switch` statement of C, C++, and Java. In those languages, the `case` expressions must be compile-time constants of the same type, and `switch` statements can often compile down to highly efficient *jump tables*.

2 When we consider the `continue` statement in §5.5.3, we’ll see that this `while` loop is not an exact equivalent of the `for` loop.