You Don't Know JS Yet: Types & Grammar - 2nd Edition

Chapter 5: Grammar

NOTE: |
:— |
Work in progress |

.

.

.

.

.

.

.

NOTE: |
:— |
Everything below here is previous text from 1st edition, and is only here for reference while 2nd edition work is underway. **Please ignore this stuff.** |

The last major topic we want to tackle is how JavaScript's language syntax works (aka its grammar). You may think you know how to write JS, but there's an awful lot of nuance to various parts of the language grammar that lead to confusion and misconception, so we want to dive into those parts and clear some things up.

Note: The term "grammar" may be a little less familiar to readers than the term "syntax." In many ways, they are similar terms, describing the *rules* for how the language works. There are nuanced differences, but they mostly don't matter for our discussion here. The grammar for JavaScript is a structured way to describe how the syntax (operators, keywords, etc.) fits together into well-formed, valid programs. In other words, discussing syntax without grammar would leave out a lot of the important details. So our focus here in this chapter is most accurately described as *grammar*, even though the raw syntax of the language is what developers directly interact with.

Statements & Expressions

It's fairly common for developers to assume that the term "statement" and "expression" are roughly equivalent. But here we need to distinguish between the two, because there are some very important differences in our JS programs.

To draw the distinction, let's borrow from terminology you may be more familiar with: the English language.

A "sentence" is one complete formation of words that expresses a thought. It's comprised of one or more "phrases," each of which can be connected with punctuation marks or conjunction words ("and," "or," etc). A phrase can itself be made up of smaller phrases. Some phrases are incomplete and don't accomplish much by themselves, while other phrases can stand on their own. These rules are collectively called the *grammar* of the English language.

And so it goes with JavaScript grammar. Statements are sentences, expressions are phrases, and operators are conjunctions/punctuation.

Every expression in JS can be evaluated down to a single, specific value result. For example:

```
var a = 3 * 6;
var b = a;
b;
```

In this snippet, `3 * 6` is an expression (evaluates to the value `18`). But `a` on the second line is also an expression, as is `b` on the third line. The `a` and `b` expressions both evaluate to the values stored in those variables at that moment, which also happens to be `18`.

Moreover, each of the three lines is a statement containing expressions. `var a = 3 * 6` and `var b = a` are called "declaration statements" because they each declare a variable (and optionally assign a value to it). The `a = 3 * 6` and `b = a` assignments (minus the `var`s) are called assignment expressions.

The third line contains just the expression `b`, but it's also a statement all by itself (though not a terribly interesting one!). This is generally referred to as an "expression statement."

Statement Completion Values

It's a fairly little known fact that statements all have completion values (even if that value is just `undefined`).

How would you even go about seeing the completion value of a statement?

The most obvious answer is to type the statement into your browser's developer console, because when you execute it, the console by default reports the completion value of the most recent statement it executed.

Let's consider `var b = a`. What's the completion value of that statement?

The `b = a` assignment expression results in the value that was assigned (`18` above), but the `var` statement itself results in `undefined`. Why? Because `var` statements are defined that way in the spec. If you put `var a = 42;` into your console, you'll see `undefined` reported back instead of `42`.

Note: Technically, it's a little more complex than that. In the ES5 spec, section 12.2 "Variable Statement," the `VariableDeclaration` algorithm actually *does* return a value (a `string` containing the name of the variable declared – weird, huh!?), but that value is basically swallowed up (except for use by the `for..in` loop) by the `VariableStatement` algorithm, which forces an empty (aka `undefined`) completion value.

In fact, if you've done much code experimenting in your console (or in a JavaScript environment REPL – read/evaluate/print/loop tool), you've probably seen `undefined` reported after many different statements, and perhaps never realized why or what that was. Put simply, the console is just reporting the statement's completion value.

But what the console prints out for the completion value isn't something we can use inside our program. So how can we capture the completion value?

That's a much more complicated task. Before we explain *how*, let's explore *why* you would want to do that.

We need to consider other types of statement completion values. For example, any regular `{ .. }` block has a completion value of the completion value of its last contained statement/expression.

Consider:

```
var b;

if (true) {
    b = 4 + 38;
}
```

If you typed that into your console/REPL, you'd probably see `42` reported, since `42` is the completion value of the `if` block, which took on the completion value of its last assignment expression statement `b = 4 + 38`.

In other words, the completion value of a block is like an *implicit return* of the last statement value in the block.

Note: This is conceptually familiar in languages like CoffeeScript, which have implicit `return` values from `function`s that are the same as the last statement value in the function.

But there's an obvious problem. This kind of code doesn't work:

```
var a, b;

a = if (true) {
    b = 4 + 38;
};
```

We can't capture the completion value of a statement and assign it into another variable in any easy syntactic/grammatical way (at least not yet!).

So, what can we do?

Warning: For demo purposes only – don't actually do the following in your real code!

We could use the much maligned `eval(..)` (sometimes pronounced "evil") function to capture this completion value.

```
var a, b;

a = eval( "if (true) { b = 4 + 38; }" );

a;  // 42
```

Yeeeaaahhhh. That's terribly ugly. But it works! And it illustrates the point that statement completion values are a real thing that can be captured not just in our console but in our programs.

There's a proposal for ES7 called "do expression." Here's how it might work:

```
var a, b;

a = do {
    if (true) {
        b = 4 + 38;
    }
};

a;  // 42
```

The `do { .. }` expression executes a block (with one or many statements in it), and the final statement completion value inside the block becomes the completion value *of* the `do` expression, which can then be assigned to `a` as shown.

The general idea is to be able to treat statements as expressions – they can show up inside other statements – without needing to wrap them in an inline function expression and perform an explicit `return ...`.

For now, statement completion values are not much more than trivia. But they're probably going to take on more significance as JS evolves, and hopefully `do { .. }` expressions will reduce the temptation to use stuff like `eval(..)`.

Warning: Repeating my earlier admonition: avoid `eval(..)`. Seriously. See the *Scope & Closures* title of this series for more explanation.

Expression Side Effects

Most expressions don't have side effects. For example:

```
var a = 2;
var b = a + 3;
```

The expression `a + 3` did not *itself* have a side effect, like for instance changing `a`. It had a result, which is `5`, and that result was assigned to `b` in the statement `b = a + 3`.

The most common example of an expression with (possible) side effects is a function call expression:

```
function foo() {
    a = a + 1;
}

var a = 1;
foo();      // result: ‘undefined‘, side effect: changed ‘a‘
```

There are other side-effecting expressions, though. For example:

```
var a = 42;
var b = a++;
```

The expression `a++` has two separate behaviors. *First*, it returns the current value of `a`, which is `42` (which then gets assigned to `b`). But *next*, it changes the value of `a` itself, incrementing it by one.

```
var a = 42;
var b = a++;

a;  // 43
b;  // 42
```

Many developers would mistakenly believe that `b` has value `43` just like `a` does. But the confusion comes from not fully considering the *when* of the side effects of the `++` operator.

The `++` increment operator and the `--` decrement operator are both unary operators (see Chapter 4), which can be used in either a postfix ("after") position or prefix ("before") position.

```
var a = 42;

a++;    // 42
a;      // 43

++a;    // 44
a;      // 44
```

When `++` is used in the prefix position as `++a`, its side effect (incrementing `a`) happens *before* the value is returned from the expression, rather than *after* as with `a++`.

Note: Would you think `++a++` was legal syntax? If you try it, you'll get a `ReferenceError` error, but why? Because side-effecting operators **require a variable reference** to target their side effects to. For `++a++`, the `a++` part is evaluated first (because of operator precedence – see below), which gives back the value of `a` *before* the increment. But then it tries to evaluate `++42`, which (if you try it) gives the same `ReferenceError` error, since `++` can't have a side effect directly on a value like `42`.

It is sometimes mistakenly thought that you can encapsulate the *after* side effect of `a++` by wrapping it in a `( )` pair, like:

```
var a = 42;
var b = (a++);

a;  // 43
b;  // 42
```

Unfortunately, `( )` itself doesn't define a new wrapped expression that would be evaluated *after* the *after side effect* of the `a++` expression, as we might have hoped. In fact, even if it did, `a++` returns `42` first, and unless you have another expression that reevaluates `a` after the side effect of `++`, you're not going to get `43` from that expression, so `b` will not be assigned `43`.

There's an option, though: the `,` statement-series comma operator. This operator allows you to string together multiple standalone expression statements into a single statement:

```
var a = 42, b;
b = ( a++, a );

a;  // 43
b;  // 43
```

Note: The `( .. )` around `a++, a` is required here. The reason is operator precedence, which we'll cover later in this chapter.

The expression `a++, a` means that the second `a` statement expression gets evaluated *after* the *after side effects* of the first `a++` statement expression, which means it returns the `43` value for assignment to `b`.

Another example of a side-effecting operator is `delete`. As we showed in Chapter 2, `delete` is used to remove a property from an `object` or a slot from an `array`. But it's usually just called as a standalone statement:

```
var obj = {
    a: 42
};

obj.a;          // 42
delete obj.a;   // true
obj.a;          // undefined
```

The result value of the `delete` operator is `true` if the requested operation is valid/allowable, or `false` otherwise. But the side effect of the operator is that it removes the property (or array slot).

Note: What do we mean by valid/allowable? Nonexistent properties, or properties that exist and are configurable (see Chapter 3 of the *this & Object Prototypes* title of this series) will return `true` from the `delete` operator. Otherwise, the result will be `false` or an error.

One last example of a side-effecting operator, which may at once be both obvious and nonobvious, is the `=` assignment operator.

Consider:

```
var a;

a = 42;     // 42
a;          // 42
```

It may not seem like `=` in `a = 42` is a side-effecting operator for the expression. But if we examine the result value of the `a = 42` statement, it's the value that

was just assigned (`42`), so the assignment of that same value into `a` is essentially a side effect.

Tip: The same reasoning about side effects goes for the compound-assignment operators like `+=`, `-=`, etc. For example, `a = b += 2` is processed first as `b += 2` (which is `b = b + 2`), and the result of *that* `=` assignment is then assigned to `a`.

This behavior that an assignment expression (or statement) results in the assigned value is primarily useful for chained assignments, such as:

```
var a, b, c;

a = b = c = 42;
```

Here, `c = 42` is evaluated to `42` (with the side effect of assigning `42` to `c`), then `b = 42` is evaluated to `42` (with the side effect of assigning `42` to `b`), and finally `a = 42` is evaluated (with the side effect of assigning `42` to `a`).

Warning: A common mistake developers make with chained assignments is like `var a = b = 42`. While this looks like the same thing, it's not. If that statement were to happen without there also being a separate `var b` (somewhere in the scope) to formally declare `b`, then `var a = b = 42` would not declare `b` directly. Depending on `strict` mode, that would either throw an error or create an accidental global (see the *Scope & Closures* title of this series).

Another scenario to consider:

```
function vowels(str) {
    var matches;

    if (str) {
        // pull out all the vowels
        matches = str.match( /[aeiou]/g );

        if (matches) {
            return matches;
        }
    }
}

vowels( "Hello World" ); // ["e","o","o"]
```

This works, and many developers prefer such. But using an idiom where we take advantage of the assignment side effect, we can simplify by combining the two `if` statements into one:

```
function vowels(str) {
    var matches;

    // pull out all the vowels
    if (str && (matches = str.match( /[aeiou]/g ))) {
        return matches;
    }
}

vowels( "Hello World" ); // ["e","o","o"]
```

Note: The `( .. )` around `matches = str.match..` is required. The reason is operator precedence, which we'll cover in the "Operator Precedence" section later in this chapter.

I prefer this shorter style, as I think it makes it clearer that the two conditionals are in fact related rather than separate. But as with most stylistic choices in JS, it's purely opinion which one is *better*.

Contextual Rules

There are quite a few places in the JavaScript grammar rules where the same syntax means different things depending on where/how it's used. This kind of thing can, in isolation, cause quite a bit of confusion.

We won't exhaustively list all such cases here, but just call out a few of the common ones.

`{ .. }` Curly Braces There's two main places (and more coming as JS evolves!) that a pair of `{ .. }` curly braces will show up in your code. Let's take a look at each of them.

Object Literals First, as an `object` literal:

```
// assume there's a 'bar()' function defined

var a = {
    foo: bar()
};
```

How do we know this is an `object` literal? Because the `{ .. }` pair is a value that's getting assigned to `a`.

Note: The `a` reference is called an "l-value" (aka left-hand value) since it's the target of an assignment. The `{ .. }` pair is an "r-value" (aka right-hand value) since it's used *just* as a value (in this case as the source of an assignment).

Labels What happens if we remove the `var a =` part of the above snippet?

```
// assume there's a `bar()` function defined

{
    foo: bar()
}
```

A lot of developers assume that the `{ .. }` pair is just a standalone `object` literal that doesn't get assigned anywhere. But it's actually entirely different.

Here, `{ .. }` is just a regular code block. It's not very idiomatic in JavaScript (much more so in other languages!) to have a standalone `{ .. }` block like that, but it's perfectly valid JS grammar. It can be especially helpful when combined with `let` block-scoping declarations (see the *Scope & Closures* title in this series).

The `{ .. }` code block here is functionally pretty much identical to the code block being attached to some statement, like a `for`/`while` loop, `if` conditional, etc.

But if it's a normal block of code, what's that bizarre looking `foo: bar()` syntax, and how is that legal?

It's because of a little known (and, frankly, discouraged) feature in JavaScript called "labeled statements." `foo` is a label for the statement `bar()` (which has omitted its trailing `;` – see "Automatic Semicolons" later in this chapter). But what's the point of a labeled statement?

If JavaScript had a `goto` statement, you'd theoretically be able to say `goto foo` and have execution jump to that location in code. `goto`s are usually considered terrible coding idioms as they make code much harder to understand (aka "spaghetti code"), so it's a *very good thing* that JavaScript doesn't have a general `goto`.

However, JS *does* support a limited, special form of `goto`: labeled jumps. Both the `continue` and `break` statements can optionally accept a specified label, in which case the program flow "jumps" kind of like a `goto`. Consider:

```
// `foo` labeled-loop
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        // whenever the loops meet, continue outer loop
        if (j == i) {
            // jump to the next iteration of
            // the `foo` labeled-loop
            continue foo;
        }
```

```
        // skip odd multiples
        if ((j * i) % 2 == 1) {
            // normal (non-labeled) `continue` of inner loop
            continue;
        }

        console.log( i, j );
    }
}
// 1 0
// 2 0
// 2 1
// 3 0
// 3 2
```

Note: `continue foo` does not mean "go to the 'foo' labeled position to continue", but rather, "continue the loop that is labeled 'foo' with its next iteration." So, it's not *really* an arbitrary `goto`.

As you can see, we skipped over the odd-multiple `3 1` iteration, but the labeled-loop jump also skipped iterations `1 1` and `2 2`.

Perhaps a slightly more useful form of the labeled jump is with `break __` from inside an inner loop where you want to break out of the outer loop. Without a labeled `break`, this same logic could sometimes be rather awkward to write:

```
// `foo` labeled-loop
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        if ((i * j) >= 3) {
            console.log( "stopping!", i, j );
            // break out of the `foo` labeled loop
            break foo;
        }

        console.log( i, j );
    }
}
// 0 0
// 0 1
// 0 2
// 0 3
// 1 0
// 1 1
// 1 2
// stopping! 1 3
```

Note: `break foo` does not mean "go to the 'foo' labeled position to continue," but rather, "break out of the loop/block that is labeled 'foo' and continue *after* it." Not exactly a `goto` in the traditional sense, huh?

The nonlabeled `break` alternative to the above would probably need to involve one or more functions, shared scope variable access, etc. It would quite likely be more confusing than labeled `break`, so here using a labeled `break` is perhaps the better option.

A label can apply to a non-loop block, but only `break` can reference such a non-loop label. You can do a labeled `break ___` out of any labeled block, but you cannot `continue ___` a non-loop label, nor can you do a non-labeled `break` out of a block.

```
function foo() {
    // `bar` labeled-block
    bar: {
        console.log( "Hello" );
        break bar;
        console.log( "never runs" );
    }
    console.log( "World" );
}

foo();
// Hello
// World
```

Labeled loops/blocks are extremely uncommon, and often frowned upon. It's best to avoid them if possible; for example using function calls instead of the loop jumps. But there are perhaps some limited cases where they might be useful. If you're going to use a labeled jump, make sure to document what you're doing with plenty of comments!

It's a very common belief that JSON is a proper subset of JS, so a string of JSON (like `{"a":42}` – notice the quotes around the property name as JSON requires!) is thought to be a valid JavaScript program. **Not true!** Try putting `{"a":42}` into your JS console, and you'll get an error.

That's because statement labels cannot have quotes around them, so `"a"` is not a valid label, and thus `:` can't come right after it.

So, JSON is truly a subset of JS syntax, but JSON is not valid JS grammar by itself.

One extremely common misconception along these lines is that if you were to load a JS file into a `<script src=..>` tag that only has JSON content in it (like from an API call), the data would be read as valid JavaScript but just be

inaccessible to the program. JSON-P (the practice of wrapping the JSON data in a function call, like `foo({"a":42})`) is usually said to solve this inaccessibility by sending the value to one of your program's functions.

Not true! The totally valid JSON value `{"a":42}` by itself would actually throw a JS error because it'd be interpreted as a statement block with an invalid label. But `foo({"a":42})` is valid JS because in it, `{"a":42}` is an `object` literal value being passed to `foo(..)`. So, properly said, **JSON-P makes JSON into valid JS grammar!**

Blocks Another commonly cited JS gotcha (related to coercion – see Chapter 4) is:

```
[] + {}; // "[object Object]"
{} + []; // 0
```

This seems to imply the `+` operator gives different results depending on whether the first operand is the `[]` or the `{}`. But that actually has nothing to do with it!

On the first line, `{}` appears in the `+` operator's expression, and is therefore interpreted as an actual value (an empty `object`). Chapter 4 explained that `[]` is coerced to `""` and thus `{}` is coerced to a `string` value as well: `"[object Object]"`.

But on the second line, `{}` is interpreted as a standalone `{}` empty block (which does nothing). Blocks don't need semicolons to terminate them, so the lack of one here isn't a problem. Finally, `+ []` is an expression that *explicitly coerces* (see Chapter 4) the `[]` to a `number`, which is the `0` value.

Object Destructuring Starting with ES6, another place that you'll see `{ .. }` pairs showing up is with "destructuring assignments" (see the *ES6 & Beyond* title of this series for more info), specifically `object` destructuring. Consider:

```
function getData() {
    // ..
    return {
        a: 42,
        b: "foo"
    };
}

var { a, b } = getData();

console.log( a, b ); // 42 "foo"
```

As you can probably tell, `var { a , b } = ..` is a form of ES6 destructuring assignment, which is roughly equivalent to:

```
var res = getData();
var a = res.a;
var b = res.b;
```

Note: `{ a, b }` is actually ES6 destructuring shorthand for `{ a: a, b: b }`, so either will work, but it's expected that the shorter `{ a, b }` will become the preferred form.

Object destructuring with a `{ .. }` pair can also be used for named function arguments, which is sugar for this same sort of implicit object property assignment:

```
function foo({ a, b, c }) {
    // no need for:
    // var a = obj.a, b = obj.b, c = obj.c
    console.log( a, b, c );
}

foo( {
    c: [1,2,3],
    a: 42,
    b: "foo"
} );    // 42 "foo" [1, 2, 3]
```

So, the context we use `{ .. }` pairs in entirely determines what they mean, which illustrates the difference between syntax and grammar. It's very important to understand these nuances to avoid unexpected interpretations by the JS engine.

`else if` And Optional Blocks It's a common misconception that JavaScript has an `else if` clause, because you can do:

```
if (a) {
    // ..
}
else if (b) {
    // ..
}
else {
    // ..
}
```

But there's a hidden characteristic of the JS grammar here: there is no `else if`. But `if` and `else` statements are allowed to omit the `{ }` around their attached block if they only contain a single statement. You've seen this many times before, undoubtedly:

```
if (a) doSomething( a );
```

Many JS style guides will insist that you always use `{ }` around a single statement block, like:

```
if (a) { doSomething( a ); }
```

However, the exact same grammar rule applies to the `else` clause, so the `else if` form you've likely always coded is *actually* parsed as:

```
if (a) {
    // ..
}
else {
    if (b) {
        // ..
    }
    else {
        // ..
    }
}
```

The `if (b) { .. } else { .. }` is a single statement that follows the `else`, so you can either put the surrounding `{ }` in or not. In other words, when you use `else if`, you're technically breaking that common style guide rule and just defining your `else` with a single `if` statement.

Of course, the `else if` idiom is extremely common and results in one less level of indentation, so it's attractive. Whichever way you do it, just call out explicitly in your own style guide/rules and don't assume things like `else if` are direct grammar rules.

Operator Precedence

As we covered in Chapter 4, JavaScript's version of `&&` and `||` are interesting in that they select and return one of their operands, rather than just resulting in `true` or `false`. That's easy to reason about if there are only two operands and one operator.

```
var a = 42;
var b = "foo";

a && b; // "foo"
a || b; // 42
```

But what about when there's two operators involved, and three operands?

```
var a = 42;
var b = "foo";
var c = [1,2,3];

a && b || c; // ???
a || b && c; // ???
```

To understand what those expressions result in, we're going to need to understand what rules govern how the operators are processed when there's more than one present in an expression.

These rules are called "operator precedence."

I bet most readers feel they have a decent grasp on operator precedence. But as with everything else we've covered in this book series, we're going to poke and prod at that understanding to see just how solid it really is, and hopefully learn a few new things along the way.

Recall the example from above:

```
var a = 42, b;
b = ( a++, a );

a;  // 43
b;  // 43
```

But what would happen if we remove the `( )`?

```
var a = 42, b;
b = a++, a;

a;  // 43
b;  // 42
```

Wait! Why did that change the value assigned to `b`?

Because the `,` operator has a lower precedence than the `=` operator. So, `b = a++, a` is interpreted as `(b = a++), a`. Because (as we explained earlier) `a++` has *after side effects*, the assigned value to `b` is the value `42` before the `++` changes `a`.

This is just a simple matter of needing to understand operator precedence. If you're going to use `,` as a statement-series operator, it's important to know that it actually has the lowest precedence. Every other operator will more tightly bind than `,` will.

Now, recall this example from above:

```
if (str && (matches = str.match( /[aeiou]/g ))) {
    // ..
}
```

We said the `( )` around the assignment is required, but why? Because `&&` has higher precedence than `=`, so without the `( )` to force the binding, the expression would instead be treated as `(str && matches) = str.match...` But this would be an error, because the result of `(str && matches)` isn't going to be a variable, but instead a value (in this case `undefined`), and so it can't be the left-hand side of an `=` assignment!

OK, so you probably think you've got this operator precedence thing down.

Let's move on to a more complex example (which we'll carry throughout the next several sections of this chapter) to *really* test your understanding:

```
var a = 42;
var b = "foo";
var c = false;

var d = a && b || c ? c || b ? a : c && b : a;

d;      // ??
```

OK, evil, I admit it. No one would write a string of expressions like that, right? *Probably* not, but we're going to use it to examine various issues around chaining multiple operators together, which *is* a very common task.

The result above is `42`. But that's not nearly as interesting as how we can figure out that answer without just plugging it into a JS program to let JavaScript sort it out.

Let's dig in.

The first question – it may not have even occurred to you to ask – is, does the first part (`a && b || c`) behave like `(a && b) || c` or like `a && (b || c)`? Do you know for certain? Can you even convince yourself they are actually different?

```
(false && true) || true;    // true
false && (true || true);    // false
```

So, there's proof they're different. But still, how does `false && true || true` behave? The answer:

```
false && true || true;      // true
(false && true) || true;    // true
```

So we have our answer. The `&&` operator is evaluated first and the `||` operator is evaluated second.

But is that just because of left-to-right processing? Let's reverse the order of operators:

```
true || false && false;     // true

(true || false) && false;   // false -- nope
true || (false && false);   // true -- winner, winner!
```

Now we've proved that `&&` is evaluated first and then `||`, and in this case that was actually counter to generally expected left-to-right processing.

So what caused the behavior? **Operator precedence**.

Every language defines its own operator precedence list. It's dismaying, though, just how uncommon it is that JS developers have read JS's list.

If you knew it well, the above examples wouldn't have tripped you up in the slightest, because you'd already know that `&&` is more precedent than `||`. But I bet a fair amount of readers had to think about it a little bit.

Note: Unfortunately, the JS spec doesn't really have its operator precedence list in a convenient, single location. You have to parse through and understand all the grammar rules. So we'll try to lay out the more common and useful bits here in a more convenient format. For a complete list of operator precedence, see "Operator Precedence" on the MDN site (* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence).

Short Circuited

In Chapter 4, we mentioned in a side note the "short circuiting" nature of operators like `&&` and `||`. Let's revisit that in more detail now.

For both `&&` and `||` operators, the right-hand operand will **not be evaluated** if the left-hand operand is sufficient to determine the outcome of the operation. Hence, the name "short circuited" (in that if possible, it will take an early shortcut out).

For example, with `a && b`, `b` is not evaluated if `a` is falsy, because the result of the `&&` operand is already certain, so there's no point in bothering to check `b`.

Likewise, with `a || b`, if `a` is truthy, the result of the operand is already certain, so there's no reason to check `b`.

This short circuiting can be very helpful and is commonly used:

```
function doSomething(opts) {
    if (opts && opts.cool) {
        // ..
    }
}
```

The `opts` part of the `opts && opts.cool` test acts as sort of a guard, because if `opts` is unset (or is not an `object`), the expression `opts.cool` would throw an error. The `opts` test failing plus the short circuiting means that `opts.cool` won't even be evaluated, thus no error!

Similarly, you can use `||` short circuiting:

```
function doSomething(opts) {
    if (opts.cache || primeCache()) {
        // ..
    }
}
```

Here, we're checking for `opts.cache` first, and if it's present, we don't call the `primeCache()` function, thus avoiding potentially unnecessary work.

Tighter Binding

But let's turn our attention back to that earlier complex statement example with all the chained operators, specifically the `? :` ternary operator parts. Does the `? :` operator have more or less precedence than the `&&` and `||` operators?

```
a && b || c ? c || b ? a : c && b : a
```

Is that more like this:

```
a && b || (c ? c || (b ? a : c) && b : a)
```

or this?

```
(a && b || c) ? (c || b) ? a : (c && b) : a
```

The answer is the second one. But why?

Because `&&` is more precedent than `||`, and `||` is more precedent than `? :`.

So, the expression `(a && b || c)` is evaluated *first* before the `? :` it participates in. Another way this is commonly explained is that `&&` and `||` "bind more tightly" than `? :`. If the reverse was true, then `c ? c...` would bind more tightly, and it would behave (as the first choice) like `a && b || (c ? c..)`.

Associativity

So, the `&&` and `||` operators bind first, then the `? :` operator. But what about multiple operators of the same precedence? Do they always process left-to-right or right-to-left?

In general, operators are either left-associative or right-associative, referring to whether **grouping happens from the left or from the right**.

It's important to note that associativity is *not* the same thing as left-to-right or right-to-left processing.

But why does it matter whether processing is left-to-right or right-to-left? Because expressions can have side effects, like for instance with function calls:

```
var a = foo() && bar();
```

Here, `foo()` is evaluated first, and then possibly `bar()` depending on the result of the `foo()` expression. That definitely could result in different program behavior than if `bar()` was called before `foo()`.

But this behavior is *just* left-to-right processing (the default behavior in JavaScript!) – it has nothing to do with the associativity of `&&`. In that example, since there's only one `&&` and thus no relevant grouping here, associativity doesn't even come into play.

But with an expression like `a && b && c`, grouping *will* happen implicitly, meaning that either `a && b` or `b && c` will be evaluated first.

Technically, `a && b && c` will be handled as `(a && b) && c`, because `&&` is left-associative (so is `||`, by the way). However, the right-associative alternative `a && (b && c)` behaves observably the same way. For the same values, the same expressions are evaluated in the same order.

Note: If hypothetically `&&` was right-associative, it would be processed the same as if you manually used `( )` to create grouping like `a && (b && c)`. But that still **doesn't mean** that `c` would be processed before `b`. Right-associativity does **not** mean right-to-left evaluation, it means right-to-left **grouping**. Either way, regardless of the grouping/associativity, the strict ordering of evaluation will be `a`, then `b`, then `c` (aka left-to-right).

So it doesn't really matter that much that `&&` and `||` are left-associative, other than to be accurate in how we discuss their definitions.

But that's not always the case. Some operators would behave very differently depending on left-associativity vs. right-associativity.

Consider the `? :` ("ternary" or "conditional") operator:

```
a ? b : c ? d : e;
```

`? :` is right-associative, so which grouping represents how it will be processed?

- `a ? b : (c ? d : e)`
- `(a ? b : c) ? d : e`

The answer is `a ? b : (c ? d : e)`. Unlike with `&&` and `||` above, the right-associativity here actually matters, as `(a ? b : c) ? d : e` *will* behave differently for some (but not all!) combinations of values.

One such example:

```
true ? false : true ? true : true;      // false

true ? false : (true ? true : true);    // false
(true ? false : true) ? true : true;    // true
```

Even more nuanced differences lurk with other value combinations, even if the end result is the same. Consider:

```
true ? false : true ? true : false;     // false

true ? false : (true ? true : false);   // false
(true ? false : true) ? true : false;   // false
```

From that scenario, the same end result implies that the grouping is moot. However:

```
var a = true, b = false, c = true, d = true, e = false;

a ? b : (c ? d : e); // false, evaluates only ‘a‘ and ‘b‘
(a ? b : c) ? d : e; // false, evaluates ‘a‘, ‘b‘ AND ‘e‘
```

So, we've clearly proved that `? :` is right-associative, and that it actually matters with respect to how the operator behaves if chained with itself.

Another example of right-associativity (grouping) is the `=` operator. Recall the chained assignment example from earlier in the chapter:

```
var a, b, c;

a = b = c = 42;
```

We asserted earlier that `a = b = c = 42` is processed by first evaluating the `c = 42` assignment, then `b = ..`, and finally `a = ...` Why? Because of the right-associativity, which actually treats the statement like this: `a = (b = (c = 42))`.

Remember our running complex assignment expression example from earlier in the chapter?

```
var a = 42;
var b = "foo";
var c = false;

var d = a && b || c ? c || b ? a : c && b : a;

d;      // 42
```

Armed with our knowledge of precedence and associativity, we should now be able to break down the code into its grouping behavior like this:

```
((a && b) || c) ? ((c || b) ? a : (c && b)) : a
```

Or, to present it indented if that's easier to understand:

```
(
  (a && b)
    ||
  c
)
  ?
(
  (c || b)
    ?
  a
    :
  (c && b)
)
  :
a
```

Let's solve it now:

1. `(a && b)` is `"foo"`.
2. `"foo" || c` is `"foo"`.
3. For the first `?` test, `"foo"` is truthy.
4. `(c || b)` is `"foo"`.
5. For the second `?` test, `"foo"` is truthy.
6. `a` is `42`.

That's it, we're done! The answer is `42`, just as we saw earlier. That actually wasn't so hard, was it?

Disambiguation

You should now have a much better grasp on operator precedence (and associativity) and feel much more comfortable understanding how code with multiple chained operators will behave.

But an important question remains: should we all write code understanding and perfectly relying on all the rules of operator precedence/associativity? Should we only use `( )` manual grouping when it's necessary to force a different processing binding/order?

Or, on the other hand, should we recognize that even though such rules *are in fact* learnable, there's enough gotchas to warrant ignoring automatic precedence/associativity? If so, should we thus always use `( )` manual grouping and remove all reliance on these automatic behaviors?

This debate is highly subjective, and heavily symmetrical to the debate in Chapter 4 over *implicit* coercion. Most developers feel the same way about both debates: either they accept both behaviors and code expecting them, or they discard both behaviors and stick to manual/explicit idioms.

Of course, I cannot answer this question definitively for the reader here anymore than I could in Chapter 4. But I've presented you the pros and cons, and hopefully encouraged enough deeper understanding that you can make informed rather than hype-driven decisions.

In my opinion, there's an important middle ground. We should mix both operator precedence/associativity *and* `( )` manual grouping into our programs – I argue the same way in Chapter 4 for healthy/safe usage of *implicit* coercion, but certainly don't endorse it exclusively without bounds.

For example, `if (a && b && c) ..` is perfectly OK to me, and I wouldn't do `if ((a && b) && c) ..` just to explicitly call out the associativity, because I think it's overly verbose.

On the other hand, if I needed to chain two `? :` conditional operators together, I'd certainly use `( )` manual grouping to make it absolutely clear what my intended logic is.

Thus, my advice here is similar to that of Chapter 4: **use operator precedence/associativity where it leads to shorter and cleaner code, but use `( )` manual grouping in places where it helps create clarity and reduce confusion.**

Automatic Semicolons

ASI (Automatic Semicolon Insertion) is when JavaScript assumes a `;` in certain places in your JS program even if you didn't put one there.

Why would it do that? Because if you omit even a single required `;` your program would fail. Not very forgiving. ASI allows JS to be tolerant of certain places where `;` aren't commonly thought to be necessary.

It's important to note that ASI will only take effect in the presence of a newline (aka line break). Semicolons are not inserted in the middle of a line.

Basically, if the JS parser parses a line where a parser error would occur (a missing expected `;`), and it can reasonably insert one, it does so. What's reasonable for insertion? Only if there's nothing but whitespace and/or comments between the end of some statement and that line's newline/line break.

Consider:

```
var a = 42, b
c;
```

Should JS treat the `c` on the next line as part of the `var` statement? It certainly would if a `,` had come anywhere (even another line) between `b` and `c`. But since there isn't one, JS assumes instead that there's an implied `;` (at the newline) after `b`. Thus, `c;` is left as a standalone expression statement.

Similarly:

```
var a = 42, b = "foo";

a
b   // "foo"
```

That's still a valid program without error, because expression statements also accept ASI.

There's certain places where ASI is helpful, like for instance:

```
var a = 42;

do {
```

```
    // ..
} while (a) // <-- ; expected here!
a;
```

The grammar requires a `;` after a `do..while` loop, but not after `while` or `for` loops. But most developers don't remember that! So, ASI helpfully steps in and inserts one.

As we said earlier in the chapter, statement blocks do not require `;` termination, so ASI isn't necessary:

```
var a = 42;

while (a) {
    // ..
} // <-- no ; expected here
a;
```

The other major case where ASI kicks in is with the `break`, `continue`, `return`, and (ES6) `yield` keywords:

```
function foo(a) {
    if (!a) return
    a *= 2;
    // ..
}
```

The `return` statement doesn't carry across the newline to the `a *= 2` expression, as ASI assumes the `;` terminating the `return` statement. Of course, `return` statements *can* easily break across multiple lines, just not when there's nothing after `return` but the newline/line break.

```
function foo(a) {
    return (
        a * 2 + 3 / 12
    );
}
```

Identical reasoning applies to `break`, `continue`, and `yield`.

Error Correction

One of the most hotly contested *religious wars* in the JS community (besides tabs vs. spaces) is whether to rely heavily/exclusively on ASI or not.

Most, but not all, semicolons are optional, but the two `;`s in the `for ( .. ) .. ` loop header are required.

On the pro side of this debate, many developers believe that ASI is a useful mechanism that allows them to write more terse (and more "beautiful") code by omitting all but the strictly required `;`s (which are very few). It is often asserted that ASI makes many `;`s optional, so a correctly written program *without them* is no different than a correctly written program *with them*.

On the con side of the debate, many other developers will assert that there are *too many* places that can be accidental gotchas, especially for newer, less experienced developers, where unintended `;`s being magically inserted change the meaning. Similarly, some developers will argue that if they omit a semicolon, it's a flat-out mistake, and they want their tools (linters, etc.) to catch it before the JS engine *corrects* the mistake under the covers.

Let me just share my perspective. A strict reading of the spec implies that ASI is an "error correction" routine. What kind of error, you may ask? Specifically, a **parser error**. In other words, in an attempt to have the parser fail less, ASI lets it be more tolerant.

But tolerant of what? In my view, the only way a **parser error** occurs is if it's given an incorrect/errored program to parse. So, while ASI is strictly correcting parser errors, the only way it can get such errors is if there were first program authoring errors – omitting semicolons where the grammar rules require them.

So, to put it more bluntly, when I hear someone claim that they want to omit "optional semicolons," my brain translates that claim to "I want to write the most parser-broken program I can that will still work."

I find that to be a ludicrous position to take and the arguments of saving keystrokes and having more "beautiful code" to be weak at best.

Furthermore, I don't agree that this is the same thing as the spaces vs tabs debate – that it's purely cosmetic – but rather I believe it's a fundamental question of writing code that adheres to grammar requirements vs. code that relies on grammar exceptions to just barely skate through.

Another way of looking at it is that relying on ASI is essentially considering newlines to be significant "whitespace." Other languages like Python have true significant whitespace. But is it really appropriate to think of JavaScript as having significant newlines as it stands today?

My take: **use semicolons wherever you know they are "required," and limit your assumptions about ASI to a minimum.**

But don't just take my word for it. Back in 2012, creator of JavaScript Brendan Eich said (http://brendaneich.com/2012/04/the-infernal-semicolon/) the following:

> The moral of this story: ASI is (formally speaking) a syntactic error correction procedure. If you start to code as if it were a universal

> significant-newline rule, you will get into trouble. .. I wish I had made newlines more significant in JS back in those ten days in May, 1995. .. Be careful not to use ASI as if it gave JS significant newlines.

Errors

Not only does JavaScript have different *subtypes* of errors (`TypeError`, `ReferenceError`, `SyntaxError`, etc.), but also the grammar defines certain errors to be enforced at compile time, as compared to all other errors that happen during runtime.

In particular, there have long been a number of specific conditions that should be caught and reported as "early errors" (during compilation). Any straight-up syntax error is an early error (e.g., `a = ,`), but also the grammar defines things that are syntactically valid but disallowed nonetheless.

Since execution of your code has not begun yet, these errors are not catchable with `try..catch`; they will just fail the parsing/compilation of your program.

Tip: There's no requirement in the spec about exactly how browsers (and developer tools) should report errors. So you may see variations across browsers in the following error examples, in what specific subtype of error is reported or what the included error message text will be.

One simple example is with syntax inside a regular expression literal. There's nothing wrong with the JS syntax here, but the invalid regex will throw an early error:

```
var a = /+foo/;     // Error!
```

The target of an assignment must be an identifier (or an ES6 destructuring expression that produces one or more identifiers), so a value like `42` in that position is illegal and can be reported right away:

```
var a;
42 = a;     // Error!
```

ES5's `strict` mode defines even more early errors. For example, in `strict` mode, function parameter names cannot be duplicated:

```
function foo(a,b,a) { }                 // just fine

function bar(a,b,a) { "use strict"; }   // Error!
```

Another `strict` mode early error is an object literal having more than one property of the same name:

```
(function(){
    "use strict";

    var a = {
        b: 42,
        b: 43
    };          // Error!
})();
```

Note: Semantically speaking, such errors aren't technically *syntax* errors but more *grammar* errors – the above snippets are syntactically valid. But since there is no `GrammarError` type, some browsers use `SyntaxError` instead.

Using Variables Too Early

ES6 defines a (frankly confusingly named) new concept called the TDZ ("Temporal Dead Zone").

The TDZ refers to places in code where a variable reference cannot yet be made, because it hasn't reached its required initialization.

The most clear example of this is with ES6 `let` block-scoping:

```
{
    a = 2;      // ReferenceError!
    let a;
}
```

The assignment `a = 2` is accessing the `a` variable (which is indeed block-scoped to the `{ .. }` block) before it's been initialized by the `let a` declaration, so it's in the TDZ for `a` and throws an error.

Interestingly, while `typeof` has an exception to be safe for undeclared variables (see Chapter 1), no such safety exception is made for TDZ references:

```
{
    typeof a;   // undefined
    typeof b;   // ReferenceError! (TDZ)
    let b;
}
```

Function Arguments

Another example of a TDZ violation can be seen with ES6 default parameter values (see the *ES6 & Beyond* title of this series):

```
var b = 3;

function foo( a = 42, b = a + b + 5 ) {
    // ..
}
```

The `b` reference in the assignment would happen in the TDZ for the parameter `b` (not pull in the outer `b` reference), so it will throw an error. However, the `a` in the assignment is fine since by that time it's past the TDZ for parameter `a`.

When using ES6's default parameter values, the default value is applied to the parameter if you either omit an argument, or you pass an `undefined` value in its place:

```
function foo( a = 42, b = a + 1 ) {
    console.log( a, b );
}

foo();                  // 42 43
foo( undefined );       // 42 43
foo( 5 );               // 5 6
foo( void 0, 7 );       // 42 7
foo( null );            // null 1
```

Note: `null` is coerced to a `0` value in the `a + 1` expression. See Chapter 4 for more info.

From the ES6 default parameter values perspective, there's no difference between omitting an argument and passing an `undefined` value. However, there is a way to detect the difference in some cases:

```
function foo( a = 42, b = a + 1 ) {
    console.log(
        arguments.length, a, b,
        arguments[0], arguments[1]
    );
}

foo();                  // 0 42 43 undefined undefined
foo( 10 );              // 1 10 11 10 undefined
foo( 10, undefined );   // 2 10 11 10 undefined
foo( 10, null );        // 2 10 null 10 null
```

Even though the default parameter values are applied to the `a` and `b` parameters, if no arguments were passed in those slots, the `arguments` array will not have entries.

Conversely, if you pass an `undefined` argument explicitly, an entry will exist in the `arguments` array for that argument, but it will be `undefined` and not (necessarily) the same as the default value that was applied to the named parameter for that same slot.

While ES6 default parameter values can create divergence between the `arguments` array slot and the corresponding named parameter variable, this same disjointedness can also occur in tricky ways in ES5:

```
function foo(a) {
    a = 42;
    console.log( arguments[0] );
}

foo( 2 );   // 42 (linked)
foo();      // undefined (not linked)
```

If you pass an argument, the `arguments` slot and the named parameter are linked to always have the same value. If you omit the argument, no such linkage occurs.

But in `strict` mode, the linkage doesn't exist regardless:

```
function foo(a) {
    "use strict";
    a = 42;
    console.log( arguments[0] );
}

foo( 2 );   // 2 (not linked)
foo();      // undefined (not linked)
```

It's almost certainly a bad idea to ever rely on any such linkage, and in fact the linkage itself is a leaky abstraction that's exposing an underlying implementation detail of the engine, rather than a properly designed feature.

Use of the `arguments` array has been deprecated (especially in favor of ES6 `...` rest parameters – see the *ES6 & Beyond* title of this series), but that doesn't mean that it's all bad.

Prior to ES6, `arguments` is the only way to get an array of all passed arguments to pass along to other functions, which turns out to be quite useful. You can also mix named parameters with the `arguments` array and be safe, as long as you follow one simple rule: **never refer to a named parameter *and* its corresponding `arguments` slot at the same time.** If you avoid that bad practice, you'll never expose the leaky linkage behavior.

```
function foo(a) {
    console.log( a + arguments[1] ); // safe!
}

foo( 10, 32 );  // 42
```

try..finally

You're probably familiar with how the `try..catch` block works. But have you ever stopped to consider the `finally` clause that can be paired with it? In fact, were you aware that `try` only requires either `catch` or `finally`, though both can be present if needed.

The code in the `finally` clause *always* runs (no matter what), and it always runs right after the `try` (and `catch` if present) finish, before any other code runs. In one sense, you can kind of think of the code in a `finally` clause as being in a callback function that will always be called regardless of how the rest of the block behaves.

So what happens if there's a `return` statement inside a `try` clause? It obviously will return a value, right? But does the calling code that receives that value run before or after the `finally`?

```
function foo() {
    try {
        return 42;
    }
    finally {
        console.log( "Hello" );
    }

    console.log( "never runs" );
}

console.log( foo() );
// Hello
// 42
```

The `return 42` runs right away, which sets up the completion value from the `foo()` call. This action completes the `try` clause and the `finally` clause immediately runs next. Only then is the `foo()` function complete, so that its completion value is returned back for the `console.log(..)` statement to use.

The exact same behavior is true of a `throw` inside `try`:

```
 function foo() {
```

```
    try {
        throw 42;
    }
    finally {
        console.log( "Hello" );
    }

    console.log( "never runs" );
}

console.log( foo() );
// Hello
// Uncaught Exception: 42
```

Now, if an exception is thrown (accidentally or intentionally) inside a `finally` clause, it will override as the primary completion of that function. If a previous `return` in the `try` block had set a completion value for the function, that value will be abandoned.

```
function foo() {
    try {
        return 42;
    }
    finally {
        throw "Oops!";
    }

    console.log( "never runs" );
}

console.log( foo() );
// Uncaught Exception: Oops!
```

It shouldn't be surprising that other nonlinear control statements like `continue` and `break` exhibit similar behavior to `return` and `throw`:

```
for (var i=0; i<10; i++) {
    try {
        continue;
    }
    finally {
        console.log( i );
    }
}
// 0 1 2 3 4 5 6 7 8 9
```

The `console.log(i)` statement runs at the end of the loop iteration, which is caused by the `continue` statement. However, it still runs before the `i++` iteration update statement, which is why the values printed are `0..9` instead of `1..10`.

Note: ES6 adds a `yield` statement, in generators (see the *Async & Performance* title of this series) which in some ways can be seen as an intermediate `return` statement. However, unlike a `return`, a `yield` isn't complete until the generator is resumed, which means a `try { .. yield .. }` has not completed. So an attached `finally` clause will not run right after the `yield` like it does with `return`.

A `return` inside a `finally` has the special ability to override a previous `return` from the `try` or `catch` clause, but only if `return` is explicitly called:

```
function foo() {
    try {
        return 42;
    }
    finally {
        // no ‘return ..‘ here, so no override
    }
}

function bar() {
    try {
        return 42;
    }
    finally {
        // override previous ‘return 42‘
        return;
    }
}

function baz() {
    try {
        return 42;
    }
    finally {
        // override previous ‘return 42‘
        return "Hello";
    }
}

foo();  // 42
bar();  // undefined
baz();  // "Hello"
```

Normally, the omission of `return` in a function is the same as `return;` or even `return undefined;`, but inside a `finally` block the omission of `return` does not act like an overriding `return undefined`; it just lets the previous `return` stand.

In fact, we can really up the craziness if we combine `finally` with labeled `break` (discussed earlier in the chapter):

```
function foo() {
    bar: {
        try {
            return 42;
        }
        finally {
            // break out of `bar` labeled block
            break bar;
        }
    }

    console.log( "Crazy" );

    return "Hello";
}

console.log( foo() );
// Crazy
// Hello
```

But... don't do this. Seriously. Using a `finally` + labeled `break` to effectively cancel a `return` is doing your best to create the most confusing code possible. I'd wager no amount of comments will redeem this code.

`switch`

Let's briefly explore the `switch` statement, a sort-of syntactic shorthand for an `if..else if..else..` statement chain.

```
switch (a) {
    case 2:
        // do something
        break;
    case 42:
        // do another thing
        break;
    default:
```

```
        // fallback to here
}
```

As you can see, it evaluates `a` once, then matches the resulting value to each `case` expression (just simple value expressions here). If a match is found, execution will begin in that matched `case`, and will either go until a `break` is encountered or until the end of the `switch` block is found.

That much may not surprise you, but there are several quirks about `switch` you may not have noticed before.

First, the matching that occurs between the `a` expression and each `case` expression is identical to the `===` algorithm (see Chapter 4). Often times `switch`es are used with absolute values in `case` statements, as shown above, so strict matching is appropriate.

However, you may wish to allow coercive equality (aka `==`, see Chapter 4), and to do so you'll need to sort of "hack" the `switch` statement a bit:

```
var a = "42";

switch (true) {
    case a == 10:
        console.log( "10 or '10'" );
        break;
    case a == 42:
        console.log( "42 or '42'" );
        break;
    default:
        // never gets here
}
// 42 or '42'
```

This works because the `case` clause can have any expression (not just simple values), which means it will strictly match that expression's result to the test expression (`true`). Since `a == 42` results in `true` here, the match is made.

Despite `==`, the `switch` matching itself is still strict, between `true` and `true` here. If the `case` expression resulted in something that was truthy but not strictly `true` (see Chapter 4), it wouldn't work. This can bite you if you're for instance using a "logical operator" like `||` or `&&` in your expression:

```
var a = "hello world";
var b = 10;

switch (true) {
    case (a || b == 10):
```

```
        // never gets here
        break;
    default:
        console.log( "Oops" );
}
// Oops
```

Since the result of `(a || b == 10)` is `"hello world"` and not `true`, the strict match fails. In this case, the fix is to force the expression explicitly to be a `true` or `false`, such as `case !!(a || b == 10):` (see Chapter 4).

Lastly, the `default` clause is optional, and it doesn't necessarily have to come at the end (although that's the strong convention). Even in the `default` clause, the same rules apply about encountering a `break` or not:

```
var a = 10;

switch (a) {
    case 1:
    case 2:
        // never gets here
    default:
        console.log( "default" );
    case 3:
        console.log( "3" );
        break;
    case 4:
        console.log( "4" );
}
// default
// 3
```

Note: As discussed previously about labeled `break`s, the `break` inside a `case` clause can also be labeled.

The way this snippet processes is that it passes through all the `case` clause matching first, finds no match, then goes back up to the `default` clause and starts executing. Since there's no `break` there, it continues executing in the already skipped over `case 3` block, before stopping once it hits that `break`.

While this sort of round-about logic is clearly possible in JavaScript, there's almost no chance that it's going to make for reasonable or understandable code. Be very skeptical if you find yourself wanting to create such circular logic flow, and if you really do, make sure you include plenty of code comments to explain what you're up to!

Review

JavaScript grammar has plenty of nuance that we as developers should spend a little more time paying closer attention to than we typically do. A little bit of effort goes a long way to solidifying your deeper knowledge of the language.

Statements and expressions have analogs in English language – statements are like sentences and expressions are like phrases. Expressions can be pure/self-contained, or they can have side effects.

The JavaScript grammar layers semantic usage rules (aka context) on top of the pure syntax. For example, `{ }` pairs used in various places in your program can mean statement blocks, `object` literals, (ES6) destructuring assignments, or (ES6) named function arguments.

JavaScript operators all have well-defined rules for precedence (which ones bind first before others) and associativity (how multiple operator expressions are implicitly grouped). Once you learn these rules, it's up to you to decide if precedence/associativity are *too implicit* for their own good, or if they will aid in writing shorter, clearer code.

ASI (Automatic Semicolon Insertion) is a parser-error-correction mechanism built into the JS engine, which allows it under certain circumstances to insert an assumed `;` in places where it is required, was omitted, *and* where insertion fixes the parser error. The debate rages over whether this behavior implies that most `;` are optional (and can/should be omitted for cleaner code) or whether it means that omitting them is making mistakes that the JS engine merely cleans up for you.

JavaScript has several types of errors, but it's less known that it has two classifications for errors: "early" (compiler thrown, uncatchable) and "runtime" (`try..catch`able). All syntax errors are obviously early errors that stop the program before it runs, but there are others, too.

Function arguments have an interesting relationship to their formal declared named parameters. Specifically, the `arguments` array has a number of gotchas of leaky abstraction behavior if you're not careful. Avoid `arguments` if you can, but if you must use it, by all means avoid using the positional slot in `arguments` at the same time as using a named parameter for that same argument.

The `finally` clause attached to a `try` (or `try..catch`) offers some very interesting quirks in terms of execution processing order. Some of these quirks can be helpful, but it's possible to create lots of confusion, especially if combined with labeled blocks. As always, use `finally` to make code better and clearer, not more clever or confusing.

The `switch` offers some nice shorthand for `if..else if..` statements, but beware of many common simplifying assumptions about its behavior. There are several quirks that can trip you up if you're not careful, but there's also some neat hidden tricks that `switch` has up its sleeve!