Appendix B: Understanding ECMAScript 7 (2016)

The development of ECMAScript 6 took about four years, and after that, TC-39 decided that such a long development process was unsustainable. Instead, they moved to a yearly release cycle to ensure new language features would make it into development sooner.

More frequent releases mean that each new edition of ECMAScript should have fewer new features than ECMAScript 6. To signify this change, new versions of the specification no longer prominently feature the edition number, and instead refer to the year in which the specification was published. As a result, ECMAScript 6 is also known as ECMAScript 2015, and ECMAScript 7 is formally known as ECMAScript 2016. TC-39 expects to use the year-based naming system for all future ECMAScript editions.

ECMAScript 2016 was finalized in March 2016 and contained only three additions to the language: a new mathematical operator, a new array method, and a new syntax error. Both are covered in this appendix.

The Exponentiation Operator

The only change to JavaScript syntax introduced in ECMAScript 2016 is the *exponentiation operator*, which is a mathematical operation that applies an exponent to a base. JavaScript already had the `Math.pow()` method to perform exponentiation, but JavaScript was also one of the only languages that required a method rather than a formal operator. (And some developers argue an operator is easier to read and reason about.)

The exponentiation operator is two asterisks (`**`) where the left operand is the base and the right operand is the exponent. For example:

```
let result = 5 ** 2;

console.log(result);                        // 25
console.log(result === Math.pow(5, 2));     // true
```

This example calculates 5^2^, which is equal to 25. You can still use `Math.pow()` to achieve the same result.

Order of Operations

The exponentiation operator has the highest precedence of all binary operators in JavaScript (unary operators have higher precedence than `**`). That means it is applied first to any compound operation, as in this example:

```
let result = 2 * 5 ** 2;
console.log(result);        // 50
```

The calculation of 5^2^ happens first. The resulting value is then multiplied by 2 for a final result of 50.

Operand Restriction

The exponentiation operator does have a somewhat unusual restriction that isn't present for other operators. The left side of an exponentiation operation cannot be a unary expression other than `++` or `--`. For example, this is invalid syntax:

```
// syntax error
let result = -5 ** 2;
```

The `-5` in this example is a syntax error because the order of operations is ambiguous. Does the `-` apply just to `5` or the result of the `5 ** 2` expression? Disallowing unary expressions on the left side of the exponentiation operator eliminates that ambiguity. In order to clearly specify intent, you need to include parentheses either around `-5` or around `5 ** 2` as follows:

```
// ok
let result1 = -(5 ** 2);    // equal to -25

// also ok
let result2 = (-5) ** 2;    // equal to 25
```

If you put the parentheses around the expression, the `-` is applied to the whole thing. When the parentheses surround `-5`, it's clear that you want to raise -5 to the second power.

You don't need parentheses to use `++` and `--` on the left side of the exponentiation operator because both operators have clearly-defined behavior on their operands. A prefix `++` or `--` changes the operand before any other operations take place, and the postfix versions don't apply any changes until after the entire expression has been evaluated. Both use cases are safe on the left side of this operator, as this code demonstrates:

```
let num1 = 2,
    num2 = 2;

console.log(++num1 ** 2);       // 9
console.log(num1);              // 3

console.log(num2-- ** 2);       // 4
console.log(num2);              // 1
```

In this example, `num1` is incremented before the exponentiation operator is applied, so `num1` becomes 3 and the result of the operation is 9. For `num2`, the value remains 2 for the exponentiation operation and then is decremented to 1.

The Array.prototype.includes() Method

You might recall that ECMAScript 6 added `String.prototype.includes()` in order to check whether certain substrings exist within a given string. Originally, ECMAScript 6 was also going to introduce an `Array.prototype.includes()` method to continue the trend of treating strings and arrays similarly. But the specification for `Array.prototype.includes()` was incomplete by the ECMAScript 6 deadline, and so `Array.prototype.includes()` ended up in ECMAScript 2016 instead.

How to Use Array.prototype.includes()

The `Array.prototype.includes()` method accepts two arguments: the value to search for and an optional index from which to start the search. When the second argument is provided, `includes()` starts the match from that index. (The default starting index is `0`.) The return value is `true` if the value is found inside the array and `false` if not. For example:

```
let values = [1, 2, 3];

console.log(values.includes(1));        // true
console.log(values.includes(0));        // false

// start the search from index 2
console.log(values.includes(1, 2));     // false
```

Here, calling `values.includes()` returns `true` for the value of `1` and `false` for the value of `0` because `0` isn't in the array. When the second argument is used to start the search at index 2 (which contains the value `3`), the `values.includes()` method returns `false` because the number `1` is not found between index 2 and the end of the array.

Value Comparison

The value comparison performed by the `includes()` method uses the `===` operator with one exception: `NaN` is considered equal to `NaN` even though `NaN === NaN` evaluates to `false`. This is different than the behavior of the `indexOf()` method, which strictly uses `===` for comparison. To see the difference, consider this code:

```
let values = [1, NaN, 2];

console.log(values.indexOf(NaN));       // -1
console.log(values.includes(NaN));      // true
```

The `values.indexOf()` method returns `-1` for `NaN` even though `NaN` is contained in the `values` array. On the other hand, `values.includes()` returns `true` for `NaN` because it uses a different value comparison operator.

W> When you want to check just for the existence of a value in an array and don't need to know the index , I recommend using `includes()` because of the difference in how `NaN` is treated by the `includes()` and `indexOf()` methods. If you do need to know where in the array a value exists, then you have to use the `indexOf()` method.

Another quirk of this implementation is that `+0` and `-0` are considered to be equal. In this case, the behavior of `indexOf()` and `includes()` is the same:

```
let values = [1, +0, 2];

console.log(values.indexOf(-0));         // 1
console.log(values.includes(-0));        // true
```

Here, both `indexOf()` and `includes()` find `+0` when `-0` is passed because the two values are considered equal. Note that this is different than the behavior of the `Object.is()` method, which considers `+0` and `-0` to be different values.

Change to Function-Scoped Strict Mode

When strict mode was introduced in ECMAScript 5, the language was quite a bit simpler than it became in ECMAScript 6. Despite that, ECMAScript 6 still allowed you to specify strict mode using the `"use strict"` directive either in the global scope (which would make all code run in strict mode) or in a function scope (so only the function would run in strict mode). The latter ended up being a problem in ECMAScript 6 due to the more complex ways that parameters could be defined, specifically, with destructuring and default parameter values. To understand the problem, consider the following code:

```
function doSomething(first = this) {
    "use strict";

    return first;
}
```

Here, the named parameter `first` is assigned a default value of `this`. What would you expect the value of `first` to be? The ECMAScript 6 specification instructed JavaScript engines to treat the parameters as being run in strict mode in this case, so `this` should be equal to `undefined`. However, implementing parameters running in strict mode when `"use strict"` is present inside the function turned out to be quite difficult because parameter default values can be functions as well. This difficulty led to most JavaScript engines not implementing this feature (so `this` would be equal to the global object).

As a result of the implementation difficulty, ECMAScript 2016 makes it illegal to have a `"use strict"` directive inside of a function whose parameters are either destructured or have default values. Only *simple parameter lists*, those that don't contain destructuring or default values, are allowed when `"use strict"` is present in the body of a function. Here are some examples:

```
// okay - using simple parameter list
function okay(first, second) {
    "use strict";

    return first;
}

// syntax error
function notOkay1(first, second=first) {
    "use strict";
    return first;
}

// syntax error
function notOkay2({ first, second }) {
    "use strict";

    return first;
}
```

You can still use `"use strict"` with simple parameter lists, which is why `okay()` works as you would expect (the same as it would in ECMAScript 5). The `notOkay1()` function is a syntax error because you can no longer use `"use strict"` in functions with default parameter values. Similarly, the `notOkay2()` function is a syntax error because you can't use `"use strict"` in a function with destructured parameters.

Overall, this change removes both a point of confusion for JavaScript developers and an implementation problem for JavaScript engines.