Appendix A: Smaller Changes

Along with the major changes this book has already covered, ECMAScript 6 made several other changes that are smaller but still helpful in improving JavaScript. Those changes include making integers easier to use, adding new methods for calculations, a tweak to Unicode identifiers, and formalizing the `__proto__` property. I describe all of those in this appendix.

Working with Integers

JavaScript uses the IEEE 754 encoding system to represent both integers and floats, which has caused a lot of confusion over the years. The language takes great pains to ensure that developers don't need to worry about the details of number encoding, but problems still leak through from time to time. ECMAScript 6 seeks to address this by making integers easier to identify and work with.

Identifying Integers

First, ECMAScript 6 added the `Number.isInteger()` method, which can determine whether a value represents an integer in JavaScript. While JavaScript uses IEEE 754 to represent both types of numbers, floats and integers are stored differently. The `Number.isInteger()` method takes advantage of that, and when the method is called on a value, the JavaScript engine looks at the underlying representation of the value to determine whether that value is an integer. That means numbers that look like floats might actually be stored as integers and cause `Number.isInteger()` to return `true`. For example:

```
console.log(Number.isInteger(25));      // true
console.log(Number.isInteger(25.0));    // true
console.log(Number.isInteger(25.1));    // false
```

In this code, `Number.isInteger()` returns `true` for both `25` and `25.0` even though the latter looks like a float. Simply adding a decimal point to a number doesn't automatically make it a float in JavaScript. Since `25.0` is really just `25`, it is stored as an integer. The number `25.1`, however, is stored as a float because there is a fraction value.

Safe Integers

IEEE 754 can only accurately represent integers between -2^53^ and 2^53^, and outside this "safe" range, binary representations end up reused for multiple numeric values. That means JavaScript can only safely represent integers within the IEEE 754 range before problems become apparent. For instance, consider this code:

```
console.log(Math.pow(2, 53));      // 9007199254740992
console.log(Math.pow(2, 53) + 1);  // 9007199254740992
```

This example doesn't contain a typo, yet two different numbers are represented by the same JavaScript integer. The effect becomes more prevalent the further the value falls outside the safe range.

ECMAScript 6 introduced the `Number.isSafeInteger()` method to better identify integers that the language can accurately represent. It also added the `Number.MAX_SAFE_INTEGER` and `Number.MIN_SAFE_INTEGER` properties to represent the upper and lower bounds of the integer range, respectively. The `Number.isSafeInteger()` method ensures that a value is an integer and falls within the safe range of integer values, as in this example:

```
var inside = Number.MAX_SAFE_INTEGER,
    outside = inside + 1;

console.log(Number.isInteger(inside));          // true
console.log(Number.isSafeInteger(inside));      // true

console.log(Number.isInteger(outside));         // true
console.log(Number.isSafeInteger(outside));     // false
```

The number `inside` is the largest safe integer, so it returns `true` for both the `Number.isInteger()` and `Number.isSafeInteger()` methods. The number `outside` is the first questionable integer value, and it isn't considered safe even though it's still an integer.

Most of the time, you only want to deal with safe integers when doing integer arithmetic or comparisons in JavaScript, so using `Number.isSafeInteger()` as part of input validation is a good idea.

New Math Methods

The new emphasis on gaming and graphics that led ECMAScript 6 to include typed arrays in JavaScript also led to the realization that a JavaScript engine could do many mathematical calculations more efficiently. But optimization strategies like asm.js, which works on a subset of JavaScript to improve performance, need more information to perform calculations in the fastest way possible. For instance, knowing whether the numbers should be treated as 32-bit integers or as 64-bit floats is important for hardware-based operations, which are much faster than software-based operations.

As a result, ECMAScript 6 added several methods to the `Math` object to improve the speed of common mathematical calculations. Improving the speed of common calculations also improves the overall speed of applications that perform many calculations, such as graphics programs. The new methods are listed below:

- `Math.acosh(x)` Returns the inverse hyperbolic cosine of `x`.
- `Math.asinh(x)` Returns the inverse hyperbolic sine of `x`.
- `Math.atanh(x)` Returns the inverse hyperbolic tangent of `x`
- `Math.cbrt(x)` Returns the cubed root of `x`.
- `Math.clz32(x)` Returns the number of leading zero bits in the 32-bit integer representation of `x`.
- `Math.cosh(x)` Returns the hyperbolic cosine of `x`.
- `Math.expm1(x)` Returns the result of subtracting 1 from the exponential function of `x`
- `Math.fround(x)` Returns the nearest single-precision float of `x`.
- `Math.hypot(...values)` Returns the square root of the sum of the squares of each argument.
- `Math.imul(x, y)` Returns the result of performing true 32-bit multiplication of the two arguments.
- `Math.log1p(x)` Returns the natural logarithm of `1 + x`.
- `Math.log10(x)` Returns the base 10 logarithm of `x`.

- `Math.log2(x)` Returns the base 2 logarithm of `x`.
- `Math.sign(x)` Returns -1 if the `x` is negative, 0 if `x` is +0 or -0, or 1 if `x` is positive.
- `Math.sinh(x)` Returns the hyperbolic sine of `x`.
- `Math.tanh(x)` Returns the hyperbolic tangent of `x`.
- `Math.trunc(x)` Removes fraction digits from a float and returns an integer.

It's beyond the scope of this book to explain each new method and what it does in detail. But if your application needs to do a reasonably common calculation, be sure to check the new `Math` methods before implementing it yourself.

Unicode Identifiers

ECMAScript 6 offers better Unicode support than previous versions of JavaScript, and it also changes what characters may be used as identifiers. In ECMAScript 5, it was already possible to use Unicode escape sequences for identifiers. For example:

```
// Valid in ECMAScript 5 and 6
var \u0061 = "abc";

console.log(\u0061);     // "abc"

// equivalent to:
console.log(a);          // "abc"
```

After the `var` statement in this example, you can use either `\u0061` or `a` to access the variable. In ECMAScript 6, you can also use Unicode code point escape sequences as identifiers, like this:

```
// Valid in ECMAScript 5 and 6
var \u{61} = "abc";

console.log(\u{61});      // "abc"

// equivalent to:
 console.log(a);          // "abc"
```

This example just replaces `\u0061` with its code point equivalent. Otherwise, it does exactly the same thing as the previous example.

Additionally, ECMAScript 6 formally specifies valid identifiers in terms of Unicode Standard Annex #31: Unicode Identifier and Pattern Syntax, which gives the following rules:

1. The first character must be `$`, `_`, or any Unicode symbol with a derived core property of `ID_Start`.
2. Each subsequent character must be `$`, `_`, `\u200c` (a zero-width non-joiner), `\u200d` (a zero-width joiner), or any Unicode symbol with a derived core property of `ID_Continue`.

The `ID_Start` and `ID_Continue` derived core properties are defined in Unicode Identifier and Pattern Syntax as a way to identify symbols that are appropriate for use in identifiers such as variables and domain names. The specification is not specific to JavaScript.

Formalizing the `__proto__` Property

Even before ECMAScript 5 was finished, several JavaScript engines already implemented a custom property called `__proto__` that could be used to both get and set the `[[Prototype]]` property. Effectively, `__proto__` was an early precursor to both the `Object.getPrototypeOf()` and `Object.setPrototypeOf()` methods. Expecting all JavaScript engines to remove this property is unrealistic (there were popular JavaScript libraries making use of `__proto__`), so ECMAScript 6 also formalized the `__proto__` behavior. But the formalization appears in Appendix B of ECMA-262 along with this warning:

> These features are not considered part of the core ECMAScript language. Programmers should not use or assume the existence of these features and behaviours when writing new ECMAScript code. ECMAScript implementations are discouraged from implementing these features unless the implementation is part of a web browser or is required to run the same legacy ECMAScript code that web browsers encounter.

The ECMAScript specification recommends using `Object.getPrototypeOf()` and `Object.setPrototypeOf()` instead because `__proto__` has the following characteristics:

1. You can only specify `__proto__` once in an object literal. If you specify two `__proto__` properties, then an error is thrown. This is the only object literal property with that restriction.
2. The computed form `["__proto__"]` acts like a regular property and doesn't set or return the current object's prototype. All rules related to object literal properties apply in this form, as opposed to the non-computed form, which has exceptions.

While you should avoid using the `__proto__` property, the way the specification defined it is interesting. In ECMAScript 6 engines, `Object.prototype.__proto__` is defined as an accessor property whose `get` method calls `Object.getPrototypeOf()` and whose `set` method calls the `Object.setPrototypeOf()` method. This leaves no real difference between using `__proto__` and `Object.getPrototypeOf()`/`Object.setPrototypeOf()`, except that `__proto__` allows you to set the prototype of an object literal directly. Here's how that works:

```js
let person = {
    getGreeting() {
        return "Hello";
    }
};

let dog = {
    getGreeting() {
        return "Woof";
    }
};

// prototype is person
let friend = {
    __proto__: person
};
```

```
console.log(friend.getGreeting());                      // "Hello"
console.log(Object.getPrototypeOf(friend) === person);  // true
console.log(friend.__proto__ === person);               // true

// set prototype to dog
friend.__proto__ = dog;
console.log(friend.getGreeting());                      // "Woof"
console.log(friend.__proto__ === dog);                  // true
console.log(Object.getPrototypeOf(friend) === dog);     // true
```

Instead of calling `Object.create()` to make the `friend` object, this example creates a standard object literal that assigns a value to the `__proto__` property. When creating an object with the `Object.create()` method, on the other hand, you'd have to specify full property descriptors for any additional object properties.