Expanded Object Functionality

ECMAScript 6 focuses heavily on improving the utility of objects, which makes sense because nearly every value in JavaScript is some type of object. Additionally, the number of objects used in an average JavaScript program continues to increase as the complexity of JavaScript applications increases, meaning that programs are creating more objects all the time. With more objects comes the necessity to use them more effectively.

ECMAScript 6 improves objects in a number of ways, from simple syntax extensions to options for manipulating and interacting with them.

Object Categories

JavaScript uses a mix of terminology to describe objects found in the standard as opposed to those added by execution environments such as the browser or Node.js, and the ECMAScript 6 specification has clear definitions for each category of object. It's important to understand this terminology to have a good understanding of the language as a whole. The object categories are:

- *Ordinary objects* Have all the default internal behaviors for objects in JavaScript.
- *Exotic objects* Have internal behavior that differs from the default in some way.
- *Standard objects* Are those defined by ECMAScript 6, such as `Array`, `Date`, and so on. Standard objects may be ordinary or exotic.
- *Built-in objects* Are present in a JavaScript execution environment when a script begins to execute. All standard objects are built-in objects.

I will use these terms throughout the book to explain the various objects defined by ECMAScript 6.

Object Literal Syntax Extensions

The object literal is one of the most popular patterns in JavaScript. JSON is built upon its syntax, and it's in nearly every JavaScript file on the Internet. The object literal is so popular because it's a succinct syntax for creating objects that otherwise would take several lines of code. Luckily for developers, ECMAScript 6 makes object literals more powerful and even more succinct by extending the syntax in several ways.

Property Initializer Shorthand

In ECMAScript 5 and earlier, object literals were simply collections of name-value pairs. That meant there could be some duplication when property values are initialized. For example:

```
function createPerson(name, age) {
    return {
        name: name,
        age: age
    };
}
```

The `createPerson()` function creates an object whose property names are the same as the function parameter names. The result appears to be duplication of `name` and `age` even though one is the name of an object property while the other provides the value for that property. The key `name` in the returned object is assigned the value contained in the variable `name`, and the key `age` in the returned object is assigned the value contained in the variable `age`.

In ECMAScript 6, you can eliminate the duplication that exists around property names and local variables by using the *property initializer* shorthand. When an object property name is the same as the local variable name, you can simply include the name without a colon and value. For example, `createPerson()` can be rewritten for ECMAScript 6 as follows:

```
function createPerson(name, age) {
    return {
        name,
        age
    };
}
```

When a property in an object literal only has a name, the JavaScript engine looks into the surrounding scope for a variable of the same name. If it finds one, that variable's value is assigned to the same name on the object literal. In this example, the object literal property `name` is assigned the value of the local variable `name`.

This extension makes object literal initialization even more succinct and helps to eliminate naming errors. Assigning a property with the same name as a local variable is a very common pattern in JavaScript, making this extension a welcome addition.

Concise Methods

ECMAScript 6 also improves the syntax for assigning methods to object literals. In ECMAScript 5 and earlier, you must specify a name and then the full function definition to add a method to an object, as follows:

```
var person = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};
```

In ECMAScript 6, the syntax is made more concise by eliminating the colon and the `function` keyword. That means you can rewrite the previous example like this:

```
var person = {
    name: "Nicholas",
    sayName() {
        console.log(this.name);
```

```
    }
};
```

This shorthand syntax, also called *concise method* syntax, creates a method on the `person` object just as the previous example did. The `sayName()` property is assigned an anonymous function and has all the same characteristics as the ECMAScript 5 `sayName()` function. The one difference is that concise methods may use `super` (discussed later in the "Easy Prototype Access with Super References" section), while the nonconcise methods may not.

I> The `name` property of a method created using concise method shorthand is the name used before the parentheses. In the last example, the `name` property for `person.sayName()` is `"sayName"`.

Computed Property Names

ECMAScript 5 and earlier could compute property names on object instances when those properties were set with square brackets instead of dot notation. The square brackets allow you to specify property names using variables and string literals that may contain characters that would cause a syntax error if used in an identifier. Here's an example:

```
var person = {},
    lastName = "last name";

person["first name"] = "Nicholas";
person[lastName] = "Zakas";

console.log(person["first name"]);      // "Nicholas"
console.log(person[lastName]);          // "Zakas"
```

Since `lastName` is assigned a value of `"last name"`, both property names in this example use a space, making it impossible to reference them using dot notation. However, bracket notation allows any string value to be used as a property name, so assigning `"first name"` to `"Nicholas"` and `"last name"` to `"Zakas"` works.

Additionally, you can use string literals directly as property names in object literals, like this:

```
var person = {
    "first name": "Nicholas"
};

console.log(person["first name"]);      // "Nicholas"
```

This pattern works for property names that are known ahead of time and can be represented with a string literal. If, however, the property name `"first name"` were contained in a variable (as in the previous example) or had to be calculated, then there would be no way to define that property using an object literal in ECMAScript 5.

In ECMAScript 6, computed property names are part of the object literal syntax, and they use the same square bracket notation that has been used to reference computed property names in object instances. For example:

```
var lastName = "last name";

var person = {
    "first name": "Nicholas",
    [lastName]: "Zakas"
};

console.log(person["first name"]);      // "Nicholas"
console.log(person[lastName]);          // "Zakas"
```

The square brackets inside the object literal indicate that the property name is computed, so its contents are evaluated as a string. That means you can also include expressions such as:

```
var suffix = " name";

var person = {
    ["first" + suffix]: "Nicholas",
    ["last" + suffix]: "Zakas"
};

console.log(person["first name"]);      // "Nicholas"
console.log(person["last name"]);       // "Zakas"
```

These properties evaluate to `"first name"` and `"last name"`, and those strings can be used to reference the properties later. Anything you would put inside square brackets while using bracket notation on object instances will also work for computed property names inside object literals.

New Methods

One of the design goals of ECMAScript beginning with ECMAScript 5 was to avoid creating new global functions or methods on `Object.prototype`, and instead try to find objects on which new methods should be available. As a result, the `Object` global has received an increasing number of methods when no other objects are more appropriate. ECMAScript 6 introduces a couple new methods on the `Object` global that are designed to make certain tasks easier.

The Object.is() Method

When you want to compare two values in JavaScript, you're probably used to using either the equals operator (`==`) or the identically equals operator (`===`). Many developers prefer the latter, to avoid type coercion during comparison. But even the identically equals operator isn't entirely accurate. For example, the values +0 and -0 are considered equal by `===` even though they are represented differently in the JavaScript engine. Also `NaN === NaN` returns `false`, which necessitates using `isNaN()` to detect `NaN` properly.

ECMAScript 6 introduces the `Object.is()` method to make up for the remaining quirks of the identically equals operator. This method accepts two arguments and returns `true` if the values are equivalent. Two values are considered equivalent when they are of the same type and have the same value. Here are some examples:

```
console.log(+0 == -0);              // true
console.log(+0 === -0);             // true
console.log(Object.is(+0, -0));     // false

console.log(NaN == NaN);            // false
console.log(NaN === NaN);           // false
console.log(Object.is(NaN, NaN));   // true

console.log(5 == 5);                // true
console.log(5 == "5");              // true
console.log(5 === 5);               // true
console.log(5 === "5");             // false
console.log(Object.is(5, 5));       // true
console.log(Object.is(5, "5"));     // false
```

In many cases, `Object.is()` works the same as the `===` operator. The only differences are that +0 and -0 are considered not equivalent and `NaN` is considered equivalent to `NaN`. But there's no need to stop using equality operators altogether. Choose whether to use `Object.is()` instead of `==` or `===` based on how those special cases affect your code.

The Object.assign() Method

Mixins are among the most popular patterns for object composition in JavaScript. In a mixin, one object receives properties and methods from another object. Many JavaScript libraries have a mixin method similar to this:

```
function mixin(receiver, supplier) {
    Object.keys(supplier).forEach(function(key) {
        receiver[key] = supplier[key];
    });

    return receiver;
}
```

The `mixin()` function iterates over the own properties of `supplier` and copies them onto `receiver` (a shallow copy, where object references are shared when property values are objects). This allows the `receiver` to gain new properties without inheritance, as in this code:

```
function EventTarget() { /*...*/ }
EventTarget.prototype = {
    constructor: EventTarget,
    emit: function() { /*...*/ },
    on: function() { /*...*/ }
```

```
};

var myObject = {};
mixin(myObject, EventTarget.prototype);

myObject.emit("somethingChanged");
```

Here, `myObject` receives behavior from the `EventTarget.prototype` object. This gives `myObject` the ability to publish events and subscribe to them using the `emit()` and `on()` methods, respectively.

This pattern became popular enough that ECMAScript 6 added the `Object.assign()` method, which behaves the same way, accepting a receiver and any number of suppliers, and then returning the receiver. The name change from `mixin()` to `assign()` reflects the actual operation that occurs. Since the `mixin()` function uses the assignment operator (`=`), it cannot copy accessor properties to the receiver as accessor properties. The name `Object.assign()` was chosen to reflect this distinction.

I> Similar methods in various libraries may have other names for the same basic functionality; popular alternates include the `extend()` and `mix()` methods. There was also, briefly, an `Object.mixin()` method in ECMAScript 6 in addition to the `Object.assign()` method. The primary difference was that `Object.mixin()` also copied over accessor properties, but the method was removed due to concerns over the use of `super` (discussed in the "Easy Prototype Access with Super References" section of this chapter).

You can use `Object.assign()` anywhere the `mixin()` function would have been used. Here's an example:

```
function EventTarget() { /*...*/ }
EventTarget.prototype = {
    constructor: EventTarget,
    emit: function() { /*...*/ },
    on: function() { /*...*/ }
}

var myObject = {}
Object.assign(myObject, EventTarget.prototype);

myObject.emit("somethingChanged");
```

The `Object.assign()` method accepts any number of suppliers, and the receiver receives the properties in the order in which the suppliers are specified. That means the second supplier might overwrite a value from the first supplier on the receiver, which is what happens in this snippet:

```
var receiver = {};

Object.assign(receiver,
    {
        type: "js",
        name: "file.js"
    },
    {
```

```
        type: "css"
    }
);

console.log(receiver.type);     // "css"
console.log(receiver.name);     // "file.js"
```

The value of `receiver.type` is `"css"` because the second supplier overwrote the value of the first.

The `Object.assign()` method isn't a big addition to ECMAScript 6, but it does formalize a common function found in many JavaScript libraries.

A> ### Working with Accessor Properties A> A> Keep in mind that `Object.assign()` doesn't create accessor properties on the receiver when a supplier has accessor properties. Since `Object.assign()` uses the assignment operator, an accessor property on a supplier will become a data property on the receiver. For example: A> A> `js A> var receiver = {}, A> supplier = { A> get name() { A> return "file.js" A> } A> }; A> A> Object.assign(receiver, supplier); A> A> var descriptor = Object.getOwnPropertyDescriptor(receiver, "name"); A> A> console.log(descriptor.value); // "file.js" A> console.log(descriptor.get); // undefined A>` A> A> In this code, the `supplier` has an accessor property called `name`. After using the `Object.assign()` method, `receiver.name` exists as a data property with a value of `"file.js"` because `supplier.name` returned `"file.js"` when `Object.assign()` was called.

Duplicate Object Literal Properties

ECMAScript 5 strict mode introduced a check for duplicate object literal properties that would throw an error if a duplicate was found. For example, this code was problematic:

```
"use strict";

var person = {
    name: "Nicholas",
    name: "Greg"        // syntax error in ES5 strict mode
};
```

When running in ECMAScript 5 strict mode, the second `name` property causes a syntax error. But in ECMAScript 6, the duplicate property check was removed. Both strict and nonstrict mode code no longer check for duplicate properties. Instead, the last property of the given name becomes the property's actual value, as shown here:

```
"use strict";

var person = {
    name: "Nicholas",
    name: "Greg"        // no error in ES6 strict mode
};
```

```
console.log(person.name);       // "Greg"
```

In this example, the value of `person.name` is `"Greg"` because that's the last value assigned to the property.

Own Property Enumeration Order

ECMAScript 5 didn't define the enumeration order of object properties, as it left this up to the JavaScript engine vendors. However, ECMAScript 6 strictly defines the order in which own properties must be returned when they are enumerated. This affects how properties are returned using `Object.getOwnPropertyNames()` and `Reflect.ownKeys` (covered in Chapter 12). It also affects the order in which properties are processed by `Object.assign()`.

The basic order for own property enumeration is:

1. All numeric keys in ascending order
2. All string keys in the order in which they were added to the object
3. All symbol keys (covered in Chapter 6) in the order in which they were added to the object

Here's an example:

```
var obj = {
    a: 1,
    0: 1,
    c: 1,
    2: 1,
    b: 1,
    1: 1
};

obj.d = 1;

console.log(Object.getOwnPropertyNames(obj).join(""));     // "012acbd"
```

The `Object.getOwnPropertyNames()` method returns the properties in `obj` in the order `0`, `1`, `2`, `a`, `c`, `b`, `d`. Note that the numeric keys are grouped together and sorted, even though they appear out of order in the object literal. The string keys come after the numeric keys and appear in the order that they were added to `obj`. The keys in the object literal itself come first, followed by any dynamic keys that were added later (in this case, `d`).

W> The `for-in` loop still has an unspecified enumeration order because not all JavaScript engines implement it the same way. The `Object.keys()` method and `JSON.stringify()` are both specified to use the same (unspecified) enumeration order as `for-in`.

While enumeration order is a subtle change to how JavaScript works, it's not uncommon to find programs that rely on a specific enumeration order to work correctly. ECMAScript 6, by defining the enumeration order, ensures that JavaScript code relying on enumeration will work correctly regardless of where it is executed.

More Powerful Prototypes

Prototypes are the foundation of inheritance in JavaScript, and ECMAScript 6 continues to make prototypes more powerful. Early versions of JavaScript severely limited what could be done with prototypes. However, as the language matured and developers became more familiar with how prototypes work, it became clear that developers wanted more control over prototypes and easier ways to work with them. As a result, ECMAScript 6 introduced some improvements to prototypes.

Changing an Object's Prototype

Normally, the prototype of an object is specified when the object is created, via either a constructor or the `Object.create()` method. The idea that an object's prototype remains unchanged after instantiation was one of the biggest assumptions in JavaScript programming through ECMAScript 5. ECMAScript 5 did add the `Object.getPrototypeOf()` method for retrieving the prototype of any given object, but it still lacked a standard way to change an object's prototype after instantiation.

ECMAScript 6 changes that assumption by adding the `Object.setPrototypeOf()` method, which allows you to change the prototype of any given object. The `Object.setPrototypeOf()` method accepts two arguments: the object whose prototype should change and the object that should become the first argument's prototype. For example:

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

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

// prototype is person
let friend = Object.create(person);
console.log(friend.getGreeting());                      // "Hello"
console.log(Object.getPrototypeOf(friend) === person);  // true

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

This code defines two base objects: `person` and `dog`. Both objects have a `getGreeting()` method that returns a string. The object `friend` first inherits from the `person` object, meaning that `getGreeting()` outputs `"Hello"`. When the prototype becomes the `dog` object, `friend.getGreeting()` outputs `"Woof"` because the original relationship to `person` is broken.

The actual value of an object's prototype is stored in an internal-only property called `[[Prototype]]`. The `Object.getPrototypeOf()` method returns the value stored in `[[Prototype]]` and `Object.setPrototypeOf()` changes the value stored in `[[Prototype]]`. However, these aren't the only ways to work with the value of `[[Prototype]]`.

Easy Prototype Access with Super References

As previously mentioned, prototypes are very important for JavaScript and a lot of work went into making them easier to use in ECMAScript 6. Another improvement is the introduction of `super` references, which make accessing functionality on an object's prototype easier. For example, to override a method on an object instance such that it also calls the prototype method of the same name, you'd do the following:

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

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


let friend = {
    getGreeting() {
        return Object.getPrototypeOf(this).getGreeting.call(this) + ",
hi!";
    }
};

// set prototype to person
Object.setPrototypeOf(friend, person);
console.log(friend.getGreeting());                      // "Hello, hi!"
console.log(Object.getPrototypeOf(friend) === person);  // true

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

In this example, `getGreeting()` on `friend` calls the prototype method of the same name. The `Object.getPrototypeOf()` method ensures the correct prototype is called, and then an additional string is appended to the output. The additional `.call(this)` ensures that the `this` value inside the prototype method is set correctly.

Remembering to use `Object.getPrototypeOf()` and `.call(this)` to call a method on the prototype is a bit involved, so ECMAScript 6 introduced `super`. At its simplest, `super` is a pointer to the current object's

prototype, effectively the `Object.getPrototypeOf(this)` value. Knowing that, you can simplify the `getGreeting()` method as follows:

```
let friend = {
    getGreeting() {
        // in the previous example, this is the same as:
        // Object.getPrototypeOf(this).getGreeting.call(this)
        return super.getGreeting() + ", hi!";
    }
};
```

The call to `super.getGreeting()` is the same as `Object.getPrototypeOf(this).getGreeting.call(this)` in this context. Similarly, you can call any method on an object's prototype by using a `super` reference, so long as it's inside a concise method. Attempting to use `super` outside of concise methods results in a syntax error, as in this example:

```
let friend = {
    getGreeting: function() {
        // syntax error
        return super.getGreeting() + ", hi!";
    }
};
```

This example uses a named property with a function, and the call to `super.getGreeting()` results in a syntax error because `super` is invalid in this context.

The `super` reference is really powerful when you have multiple levels of inheritance, because in that case, `Object.getPrototypeOf()` no longer works in all circumstances. For example:

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

// prototype is person
let friend = {
    getGreeting() {
        return Object.getPrototypeOf(this).getGreeting.call(this) + ",
hi!";
    }
};
Object.setPrototypeOf(friend, person);


// prototype is friend
let relative = Object.create(friend);
```

```
console.log(person.getGreeting());                  // "Hello"
console.log(friend.getGreeting());                  // "Hello, hi!"
console.log(relative.getGreeting());                // error!
```

The call to `Object.getPrototypeOf()` results in an error when `relative.getGreeting()` is called. That's because `this` is `relative`, and the prototype of `relative` is the `friend` object. When `friend.getGreeting().call()` is called with `relative` as `this`, the process starts over again and continues to call recursively until a stack overflow error occurs.

That problem is difficult to solve in ECMAScript 5, but with ECMAScript 6 and `super`, it's easy:

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

// prototype is person
let friend = {
    getGreeting() {
        return super.getGreeting() + ", hi!";
    }
};
Object.setPrototypeOf(friend, person);


// prototype is friend
let relative = Object.create(friend);

console.log(person.getGreeting());                  // "Hello"
console.log(friend.getGreeting());                  // "Hello, hi!"
console.log(relative.getGreeting());                // "Hello, hi!"
```

Because `super` references are not dynamic, they always refer to the correct object. In this case, `super.getGreeting()` always refers to `person.getGreeting()`, regardless of how many other objects inherit the method.

A Formal Method Definition

Prior to ECMAScript 6, the concept of a "method" wasn't formally defined. Methods were just object properties that contained functions instead of data. ECMAScript 6 formally defines a method as a function that has an internal `[[HomeObject]]` property containing the object to which the method belongs. Consider the following:

```
let person = {

    // method
```

```
    getGreeting() {
        return "Hello";
    }
};

// not a method
function shareGreeting() {
    return "Hi!";
}
```

This example defines `person` with a single method called `getGreeting()`. The `[[HomeObject]]` for `getGreeting()` is `person` by virtue of assigning the function directly to an object. The `shareGreeting()` function, on the other hand, has no `[[HomeObject]]` specified because it wasn't assigned to an object when it was created. In most cases, this difference isn't important, but it becomes very important when using `super` references.

Any reference to `super` uses the `[[HomeObject]]` to determine what to do. The first step is to call `Object.getPrototypeOf()` on the `[[HomeObject]]` to retrieve a reference to the prototype. Then, the prototype is searched for a function with the same name. Last, the `this` binding is set and the method is called. Here's an example:

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

// prototype is person
let friend = {
    getGreeting() {
        return super.getGreeting() + ", hi!";
    }
};
Object.setPrototypeOf(friend, person);

console.log(friend.getGreeting());  // "Hello, hi!"
```

Calling `friend.getGreeting()` returns a string, which combines the value from `person.getGreeting()` with `", hi!"`. The `[[HomeObject]]` of `friend.getGreeting()` is `friend`, and the prototype of `friend` is `person`, so `super.getGreeting()` is equivalent to `person.getGreeting.call(this)`.

Summary

Objects are the center of programming in JavaScript, and ECMAScript 6 made some helpful changes to objects that both make them easier to deal with and more powerful.

ECMAScript 6 makes several changes to object literals. Shorthand property definitions make assigning properties with the same names as in-scope variables easier. Computed property names allow you to specify non-literal values as property names, which you've already been able to do in other areas of the language.

Shorthand methods let you type a lot fewer characters in order to define methods on object literals, by completely omitting the colon and `function` keyword. ECMAScript 6 loosens the strict mode check for duplicate object literal property names as well, meaning you can have two properties with the same name in a single object literal without throwing an error.

The `Object.assign()` method makes it easier to change multiple properties on a single object at once. This can be very useful if you use the mixin pattern. The `Object.is()` method performs strict equality on any value, effectively becoming a safer version of `===` when dealing with special JavaScript values.

Enumeration order for own properties is now clearly defined in ECMAScript 6. When enumerating properties, numeric keys always come first in ascending order followed by string keys in insertion order and symbol keys in insertion order.

It's now possible to modify an object's prototype after it's already created, thanks to ECMAScript 6's `Object.setPrototypeOf()` method.

Finally, you can use the `super` keyword to call methods on an object's prototype. The `this` binding inside a method invoked using `super` is set up to automatically work with the current value of `this`.