Understanding ECMAScript 6

Table of Contents

Introduction

The JavaScript core language features are defined in a standard called ECMA-262. The language defined in this standard is called ECMAScript, of which the JavaScript in the browser and Node.js environments are a superset. While browsers and Node.js may add more capabilities through additional objects and methods, the core of the language remains as defined in ECMAScript, which is why the ongoing development of ECMA-262 is vital to the success of JavaScript as a whole.

In 2007, JavaScript was at a crossroads. The popularity of Ajax was ushering in a new age of dynamic web applications while JavaScript hadn't changed since the third edition of ECMA-262 was published in 1999. TC-39, the committee responsible for driving the ECMAScript process, put together a large draft specification for ECMAScript 4. ECMAScript 4 was massive in scope, introducing changes both small and large to the language. Language features included new syntax, modules, classes, classical inheritance, private object members, optional type annotations, and more.

The scope of the ECMAScript 4 changes caused a rift to form in TC-39, with some members feeling that the fourth edition was trying to accomplish too much. A group of leaders from Yahoo, Google, and Microsoft came up with an alternate proposal for the next version of ECMAScript that they initially called ECMAScript 3.1. The "3.1" was intended to show that this was an incremental change to the existing standard.

ECMAScript 3.1 introduced very few syntax changes, instead focusing on property attributes, native JSON support, and adding methods to already-existing objects. Although there was an early attempt to reconcile ECMAScript 3.1 and ECMAScript 4, this ultimately failed as the two camps had difficulty with the very different perspectives on how the language should grow.

In 2008, Brendan Eich, the creator of JavaScript, announced that TC-39 would focus its efforts on standardizing ECMAScript 3.1. They would table the major syntax and feature changes of ECMAScript 4 until after the next version of ECMAScript was standardized, and all members of the committee would work to bring the best pieces of ECMAScript 3.1 and 4 together after that point into an effort initially nicknamed ECMAScript Harmony.

ECMAScript 3.1 was eventually standardized as the fifth edition of ECMA-262, also described as ECMAScript 5. The committee never released an ECMAScript 4 standard to avoid confusion with the now-defunct effort of the same name. Work then began on ECMAScript Harmony, with ECMAScript 6 being the first standard released in this new "harmonious" spirit.

ECMAScript 6 reached feature complete status in 2015 was formally dubbed "ECMAScript 2015" (though this text still refers to it as ECMAScript 6, the name most familiar to developers). The features vary widely from completely new objects and patterns to syntax changes to new methods on existing objects. The exciting thing about ECMAScript 6 is that all of these changes are geared towards problems that developers are actually facing. And while it will still take time for adoption and implementation to reach the point where ECMAScript 6 is the minimum that developers can expect, there's a lot to be gained from a good understanding of what the future of JavaScript looks like.

Browser and Node.js Compatibility

Many JavaScript environments, such as web browsers and Node.js, are actively working on implementing ECMAScript 6. This book does not attempt to address the inconsistencies between implementations and instead focuses on what the specification defines as the correct behavior. As such, it's possible that your JavaScript environment may not conform to the behavior described in this book.

Who This Book is For

This book is intended as a guide for those who are already familiar with JavaScript and ECMAScript 5. While a deep understanding of the language isn't necessary to use this book, it is helpful in understanding the differences between ECMAScript 5 and 6. In particular, this book is aimed at intermediate-to-advanced JavaScript developers (both browser and Node.js environments) who want to learn about the future of the language.

This book is not for beginners who have never written JavaScript. You will need to have a good basic understanding of the language to make use of this book.

Overview

Chapter 1: Block Bindings talks about `let` and `const`, the block-level replacement for `var`.

Chapter 2: Strings and Regular Expressions covers the additions to string manipulation and inspection as well as the introduction of template strings.

Chapter 3: Functions discusses the various changes to functions. This includes the arrow function form, default parameters, rest parameters, and more.

Chapter 4: Objects explains the changes to how objects are created, modified, and used. Topics include changes to object literal syntax, and new reflection methods.

Chapter 5: Destructuring introduces object and array destructuring, which allow you to decompose objects and arrays using a concise syntax.

Chapter 6: Symbols introduces the concept of symbols, a new way to define properties. Symbols are a new primitive type that can be used to obscure (but not hide) object properties and methods.

Chapter 7: Sets and Maps details the new collection types of Set, WeakSet, Map, and WeakMap. These types expand on the usefulness of arrays by adding semantics, de-duping, and memory management designed specifically for JavaScript.

Chapter 8: Iterators and Generators discusses the addition of iterators and generators to the language. These features allow you to work with collections of data in powerful ways that were not possible in previous versions of JavaScript.

Chapter 9: Classes introduces the first formal concept of classes in JavaScript. Often a point of confusion for those coming from other languages, the addition of class syntax in JavaScript makes the language more approachable to others and more concise for enthusiasts.

Chapter 10: Arrays details the changes to native arrays and the interesting new ways they can be used in JavaScript.

Chapter 11: Promises introduces promises as a new part of the language. Promises were a grassroots effort that eventually took off and gained in popularity due to extensive library support. ECMAScript 6 formalizes promises and makes them available by default.

Chapter 12: Reflection introduces the formalized reflection API for JavaScript. Similar to other languages, ECMAScript 6 reflection allows you to inspect objects at a granular level, even if you didn’t create the object.

Chapter 13: Proxies discusses the new proxy object that allows you to intercept every operation performed on an object. Proxies give developers unprecedented control over objects and, as such, unlimited possibilities for defining new interaction patterns.

Chapter 14: Modules details the official module format for JavaScript. The intent is that these modules can replace the numerous ad-hoc module definition formats that have appeared over the years.

Help and Support

You can file issues, suggest changes, and open pull requests against this book by visiting: https://github.com/nzakas/understandinges6

For anything else, please send a message to the mailing list:

http://groups.google.com/group/zakasbooks.

Block Bindings

Traditionally, the way variable declarations work has been one tricky part of programming in JavaScript. In most C-based languages, variables (or bindings) are created at the spot where the declaration occurs. In JavaScript, however, this is not the case. Where your variables are actually created depends on how you declare them, and ECMAScript 6 offers options to make controlling scope easier. This chapter demonstrates why classic `var` declarations can be confusing, introduces block-level bindings in ECMAScript 6, and then offers some best practices for using them.

Var Declarations and Hoisting

Variable declarations using `var` are treated as if they are at the top of the function (or global scope, if declared outside of a function) regardless of where the actual declaration occurs; this is called hoisting. For a demonstration of what hoisting does, consider the following function definition:

```
function getValue(condition) {

    if (condition) {
        var value = "blue";

        // other code

        return value;
    } else {

        // value exists here with a value of undefined

        return null;
    }

        // value exists here with a value of undefined
}
```

If you are unfamiliar with JavaScript, then you might expect the variable `value` to only be created if `condition` evaluates to true. In fact, the variable `value` is created regardless. Behind the scenes, the JavaScript engine changes the `getValue` function to look like this:

```
function getValue(condition) {

    var value;

    if (condition) {
        value = "blue";

        // other code

        return value;
    } else {

        return null;
    }
}
```

The declaration of value is hoisted to the top, while the initialization remains in the same spot. That means the variable value is actually still accessible from within the else clause. If accessed from there, the variable would just have a value of undefined because it hasn't been initialized.

It often takes new JavaScript developers some time to get used to declaration hoisting, and misunderstanding this unique behavior can end up causing bugs. For this reason, ECMAScript 6 introduces block level scoping options to make the controlling a variable's lifecycle a little more powerful.

Block-Level Declarations

Block-level declarations are those that declare variables that are inaccessible outside of a given block scope. Block scopes are created:

1. Inside of a function
2. Inside of a block (indicated by the { and } characters)

Block scoping is how many C-based languages work, and the introduction of block-level declarations in ECMAScript 6 is intended to bring that same flexibility (and uniformity) to JavaScript.

Let Declarations

The let declaration syntax is the same as the syntax for var. You can basically replace var with let to declare a variable, but limit the variable's scope to only the current code block (there are a few other subtle differences discussed a bit later, as well). Since let declarations are not hoisted to the top of the enclosing block, you may want to always place let declarations first in the block, so that they are available to the entire block. Here's an example:

```
function getValue(condition) {

    if (condition) {
        let value = "blue";

        // other code

        return value;
    } else {

        // value doesn't exist here

        return null;
    }

    // value doesn't exist here
}
```

This version of the getValue function behaves much closer to how you'd expect it to in other C-based languages. Since the variable value is declared using let instead of var, the declaration isn't hoisted to the top of the function definition, and the variable value is destroyed once execution flows out of the if block. If condition evaluates to false, then value is never declared or initialized.

No Redeclaration

If an identifier has already been defined in a scope, then using the identifier in a let declaration inside that scope causes an error to be thrown. For example:

```
var count = 30;

// Syntax error
let count = 40;
```

In this example, count is declared twice: once with var and once with let. Because let will not redefine an identifier that already exists in the same scope, the let declaration will throw an error. On the other hand, no error is thrown if a let declaration creates a new variable with the same name as a variable in its containing scope, as demonstrated in the following code:

```
var count = 30;

// Does not throw an error
if (condition) {

    let count = 40;

    // more code
}
```

This let declaration doesn’t throw an error because it creates a new variable called count within the if statement, instead of creating count in the surrounding block. Inside the if block, this new variable shadows the global count, preventing access to it until execution leaves the block.

Constant Declarations

You can also define variables in ECMAScript 6 with the const declaration syntax. Variables declared using const are considered constants, meaning their values cannot be changed once set. For this reason, every const variable must be initialized on declaration, as shown in this example:

```
// Valid constant
const maxItems = 30;

// Syntax error: missing initialization
const name;
```

The maxItems variable is initialized, so its const declaration should work without a problem. The name variable, however, would cause a syntax error if you tried to run the program containing this code, because name is not initialized.

Constants vs Let Declarations

Constants, like let declarations, are block-level declarations. That means constants are destroyed once execution flows out of the block in which they were declared, and declarations are not hoisted, as demonstrated in this example:

```
if (condition) {
    const maxItems = 5;

    // more code
}

// maxItems isn't accessible here
```

In this code, the constant maxItems is declared within an if statement. Once the statement finishes executing, maxItems is destroyed and is not accessible outside of that block.

In another similarity to let, a const declaration throws an error when made with an identifier for an already-defined variable in the same scope. It doesn't matter if that variable was declared using var (for global or function scope) or let (for block scope). For example, consider this code:

```
var message = "Hello!";
let age = 25;

// Each of these would throw an error.
const message = "Goodbye!";
const age = 30;
```

The two const declarations would be valid alone, but given the previous var and let declarations in this case, neither will work as intended.

Despite those similarities, there is one big difference between let and const to remember. Attempting to assign a const to a previously defined constant will throw an error, in both strict and non-strict modes:

```
const maxItems = 5;

maxItems = 6;      // throws error
```

Much like constants in other languages, the maxItems variable can't be assigned a new value later on. However, unlike constants in other language, the value a constant holds may be modified if it is an object.

Declaring Objects with Const

A const declaration prevents modification of the binding and not of the value itself. That means const declarations for objects do not prevent modification of those objects. For example:

```
const person = {
    name: "Nicholas"
};

// works
person.name = "Greg";

// throws an error
person = {
    name: "Greg"
};
```

Here, the binding person is created with an initial value of an object with one property. It’s possible to change person.name without causing an error because this changes what person contains and doesn’t change the value that person is bound to. When this code attempts to assign a value to person (thus attempting to change the binding), an error will be thrown. This subtlety in how const works with objects is easy to misunderstand. Just remember: const prevents modification of the binding, not modification of the bound value.

⚠ Several browsers implement pre-ECMAScript 6 versions of const, so be aware of this when you use this declaration type. Implementations range from being simply a synonym for var (allowing the value to be overwritten) to actually defining constants but only in the global or function scope. For this reason, be especially careful with using const in a production system. It may not provide the functionality you expect.

The Temporal Dead Zone

Unlike var syntax, let and const variables have no hoisting characteristics. A variable declared with either cannot be accessed until after the declaration. Attempting to do so results in a reference error, even when using normally safe operations such as the typeof operation in this if statement:

```
if (condition) {
    console.log(typeof value);  // ReferenceError!
    let value = "blue";
}
```

Here, the variable value is defined and initialized using let, but that statement is never executed because the previous line throws an error. The issue is that value exists in what the JavaScript community has dubbed the temporal dead zone (TDZ). The TDZ is never named explicitly in the ECMAScript specification, but the term is often used to describe the non-hoisting behavior of let and const. This section covers some subtleties of declaration placement that the TDZ causes, and although the examples shown all use let, note that the same information applies to const.

When a JavaScript engine looks through an upcoming block and finds a variable declaration, it either hoists the declaration (for var) or places the declaration in the TDZ (for let and const). Any attempt to access a variable in the TDZ results in a runtime error. That variable is only removed from the TDZ, and therefore safe to use, once execution flows to the variable declaration.

This is true anytime you attempt to use a variable declared with let before it's been defined. As the previous example demonstrated, this even applies to the normally safe typeof operator. You can, however, use typeof on a variable outside of the block where that variable is declared, though it may not give the results you're after. Consider this code:

```
console.log(typeof value);     // "undefined"

if (condition) {
    let value = "blue";
}
```

The variable value isn't in the TDZ when the typeof operation executes because it occurs outside of the block in which value is declared. That means there is no value binding, and typeof simply returns "undefined".

The TDZ is just one unique aspect of block bindings. Another unique aspect has to do with their use inside of loops.

Block Binding in Loops

Perhaps one area where developers most want block level scoping of variables is within for loops, where the throwaway counter variable is meant to be used only inside the loop. For instance, it’s not uncommon to see code like this in JavaScript:

```
for (var i=0; i < 10; i++) {
    process(items[i]);
}

// i is still accessible here
console.log(i);                     // 10
```

In other languages, where block level scoping is the default, this example should work as intended, and only the for loop should have access to the i variable. In JavaScript, however, the variable i is still accessible after the loop is completed because the var declaration gets hoisted. Using let instead, as in the following code, should give the intended behavior:

```
for (let i=0; i < 10; i++) {
    process(items[i]);
}

// i is not accessible here - throws an error
console.log(i);
```

In this example, the variable i only exists within the for loop. Once the loop is complete, the variable is destroyed and is no longer accessible elsewhere.

Functions in Loops

The characteristics of var have long made creating functions inside of loops problematic, because the loop variables are accessible from outside the scope of the loop. Consider the following code:

```
var funcs = [];

for (var i=0; i < 10; i++) {
    funcs.push(function() { console.log(i); });
}

funcs.forEach(function(func) {
    func();     // outputs the number "10" ten times
});
```

You might ordinarily expect this code to print the numbers 0 to 9, but it outputs the number 10 ten times in a row. That’s because i is shared across each iteration of the loop, meaning the functions created inside the loop all hold a reference to the same variable. The variable i has a value of 10 once the loop completes, and so when console.log(i) is called, that value prints each time.

To fix this problem, developers use immediately-invoked function expressions (IIFEs) inside of loops to force a new copy of the variable they want to iterate over to be created, as in this example:

```
var funcs = [];

for (var i=0; i < 10; i++) {
    funcs.push((function(value) {
        return function() {
            console.log(value);
        }
    }(i)));
}

funcs.forEach(function(func) {
    func();     // outputs 0, then 1, then 2, up to 9
});
```

This version uses an IIFE inside of the loop. The i variable is passed to the IIFE, which creates its own copy and stores it as value. This is the value used by the function for that iteration, so calling each function returns the expected value as the loop counts up from 0 to 9. Fortunately, block-level binding with let and const in ECMAScript 6 can simplify this loop for you.

Let Declarations in Loops

A let declaration simplifies loops by effectively mimicking what the IIFE does in the previous example. On each iteration, the loop creates a new variable and initializes it to the value of the variable with the same name from the previous iteration. That means you can omit the IIFE altogether and get the results you expect, like this:

```
var funcs = [];

for (let i=0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}

funcs.forEach(function(func) {
    func();     // outputs 0, then 1, then 2, up to 9
})
```

This loop works exactly like the loop that used `var` and an IIFE but is, arguably, cleaner. The `let` declaration creates a new variable `i` each time through the loop, so each function created inside the loop gets its own copy of `i`. Each copy of `i` has the value it was assigned at the beginning of the loop iteration in which it was created. The same is true for `for-in` and `for-of` loops, as shown here:

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };

for (let key in object) {
    funcs.push(function() {
        console.log(key);
    });
}

funcs.forEach(function(func) {
    func();     // outputs "a", then "b", then "c"
});
```

In this example, the `for-in` loop shows the same behavior as the `for` loop. Each time through the loop, a new `key` binding is created, and so each function has its own copy of the `key` variable. The result is that each function outputs a different value. If `var` were used to declare `key`, all functions would output `"c"`.

It’s important to understand that the behavior of let declarations in loops is a specially-defined behavior in the specification and is not necessarily related to the non-hoisting characteristics of let. In fact, early implementations of let did not have this behavior, as it was added later on in the process.

Constant Declarations in Loops

The ECMAScript 6 specification doesn’t explicitly disallow const declarations in loops; however, there are different behaviors based on the type of loop you’re using. For a normal for loop, you can use const in the initializer, but the loop will throw a warning if you attempt to change the value. For example:

```
var funcs = [];

// throws an error after one iteration
for (const i=0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}
```

In this code, the i variable is declared as a constant. The first iteration of the loop, where i is 0, executes successfully. An error is thrown when i++ executes because it’s attempting to modify a constant. As such, you can only use const to declare a variable in the loop initializer if you are not modifying that variable.

When used in a for-in or for-of loop, on the other hand, a const variable behaves the same as a let variable. So the following should not cause an error:

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };

// doesn't cause an error
for (const key in object) {
    funcs.push(function() {
        console.log(key);
    });
}

funcs.forEach(function(func) {
    func();     // outputs "a", then "b", then "c"
});
```

This code functions almost exactly the same as the second example in the “Let Declarations in Loops” section. The only difference is that the value of key cannot be changed inside the loop. The for-in and for-of loops work with const because the loop initializer creates a new binding on each iteration through the loop rather than attempting to modify the value of an existing binding (as was the case with the previous example using for instead of for-in).

Global Block Bindings

Another way in which let and const are different from var is in their global scope behavior. When var is used in the global scope, it creates a new global variable, which is a property on the global object (window in browsers). That means you can accidentally overwrite an existing global using var, such as:

```
// in a browser
var RegExp = "Hello!";
console.log(window.RegExp);     // "Hello!"

var ncz = "Hi!";
console.log(window.ncz);        // "Hi!"
```

Even though the RegExp global is defined on window, it is not safe from being overwritten by a var declaration. This example declares a new global variable RegExp that overwrites the original. Similarly, ncz is defined as a global variable and immediately defined as a property on window. This is the way JavaScript has always worked.

If you instead use let or const in the global scope, a new binding is created in the global scope but no property is added to the global object. That also means you cannot overwrite a global variable using let or const, you can only shadow it. Here's an example:

```
// in a browser
let RegExp = "Hello!";
console.log(RegExp);                    // "Hello!"
console.log(window.RegExp === RegExp);  // false

const ncz = "Hi!";
console.log(ncz);                       // "Hi!"
console.log("ncz" in window);           // false
```

Here, a new let declaration for RegExp creates a binding that shadows the global RegExp. That means window.RegExp and RegExp are not the same, so there is no disruption to the global scope. Also, the const declaration for ncz creates a binding but does not create a property on the global object. This capability makes let and const a lot safer to use in the global scope when you don't want to create properties on the global object.

You may still want to use var in the global scope if you have a code that should be available from the global object. This is most common in a browser when you want to access code across frames or windows.

Emerging Best Practices for Block Bindings

While ECMAScript 6 was in development, there was widespread belief you should use let by default instead of var for variable declarations. For many JavaScript developers, let behaves exactly the way they thought var should have behaved, and so the direct replacement makes logical sense. In this case, you would use const for variables that needed modification protection.

However, as more developers migrated to ECMAScript 6, an alternate approach gained popularity: use const by default and only use let when you know a variable's value needs to change. The rationale is that most variables should not change their value after initialization because unexpected value changes are a source of bugs. This idea has a significant amount of traction and is worth exploring in your code as you adopt ECMAScript 6.

Summary

The `let` and `const` block bindings introduce lexical scoping to JavaScript. These declarations are not hoisted and only exist within the block in which they are declared. This offers behavior that is more like other languages and less likely to cause unintentional errors, as variables can now be declared exactly where they are needed. As a side effect, you cannot access variables before they are declared, even with safe operators such as `typeof`. Attempting to access a block binding before its declaration results in an error due to the binding's presence in the temporal dead zone (TDZ).

In many cases, `let` and `const` behave in a manner similar to `var`; however, this is not true for loops. For both `let` and `const`, `for-in` and `for-of` loops create a new binding with each iteration through the loop. That means functions created inside the loop body can access the loop bindings values as they are during the current iteration, rather than as they were after the loop's final iteration (the behavior with `var`). The same is true for `let` declarations in `for` loops, while attempting to use `const` declarations in a `for` loop may result in an error.

The current best practice for block bindings is to use `const` by default and only use `let` when you know a variable's value needs to change. This ensures a basic level of immutability in code that can help prevent certain types of errors.

Strings and Regular Expressions

Strings are arguably one of the most important data types in programming. They're in nearly every higher-level programming language, and being able to work with them effectively is fundamental for developers to create useful programs. By extension, regular expressions are important because of the extra power they give developers to wield on strings. With these facts in mind, the creators of ECMAScript 6 improved strings and regular expressions by adding new capabilities and long-missing functionality. This chapter gives a tour of both types of changes.

Better Unicode Support

Before ECMAScript 6, JavaScript strings revolved around 16-bit character encoding (UTF-16). Each 16-bit sequence is a code unit representing a character. All string properties and methods, like the `length` property and the charAt() method, were based on these 16-bit code units. Of course, 16 bits used to be enough to contain any character. That's no longer true thanks to the expanded character set introduced by Unicode.

UTF-16 Code Points

Limiting character length to 16 bits wasn't possible for Unicode's stated goal of providing a globally unique identifier to every character in the world. These globally unique identifiers, called code points, are simply numbers starting at 0.

Code points are like character codes, but there's a subtle difference. A character encoding translates code points into code units that are internally consistent. For UTF-16, code points can be made up of many code units.

The first 2^{16} code points in UTF-16 are represented as single 16-bit code units. This range is called the Basic Multilingual Plane (BMP). Everything beyond that is considered to be in one of the supplementary planes, where the code points can no longer be represented in just 16-bits. UTF-16 solves this problem by introducing surrogate pairs in which a single code point is represented by two 16-bit code units. That means any single character in a string can be either one code unit for BMP characters, giving a total of 16 bits, or two units for supplementary plane characters, giving a total of 32 bits.

In ECMAScript 5, all string operations work on 16-bit code units, meaning that you can get unexpected results from UTF-16 encoded strings containing surrogate pairs, as in this example:

```
var text = " ";

console.log(text.length);           // 2
console.log(/^.$/.test(text));      // false
console.log(text.charAt(0));         // ""
console.log(text.charAt(1));         // ""
console.log(text.charCodeAt(0));     // 55362
console.log(text.charCodeAt(1));     // 57271
```

The single Unicode character " " is represented using surrogate pairs, and as such, the JavaScript string operations above treat the string as having two 16-bit characters. That means:

- The length of text is 2, when it should be 1.
- A regular expression trying to match a single character fails because it thinks there are two characters.

- The charAt() method is unable to return a valid character string, because neither set of 16 bits corresponds to a printable character.

The charCodeAt() method also just can't identify the character properly. It returns the appropriate 16-bit number for each code unit, but that is the closest you could get to the real value of text in ECMAScript 5.

ECMAScript 6, on the other hand, enforces UTF-16 string encoding to address problems like these. Standardizing string operations based on this character encoding means that JavaScript can support functionality designed to work specifically with surrogate pairs. The rest of this section discusses a few key examples of that functionality.

The codePointAt() Method

One method ECMAScript 6 added to fully support UTF-16 is the codePointAt() method, which retrieves the Unicode code point that maps to a given position in a string. This method accepts the code unit position rather than the character position and returns an integer value, as these console.log() examples show:

```
var text = " a";

console.log(text.charCodeAt(0));    // 55362
console.log(text.charCodeAt(1));    // 57271
console.log(text.charCodeAt(2));    // 97

console.log(text.codePointAt(0));   // 134071
console.log(text.codePointAt(1));   // 57271
console.log(text.codePointAt(2));   // 97
```

The codePointAt() method returns the same value as the charCodeAt() method unless it operates on non-BMP characters. The first character in text is non-BMP and is therefore comprised of two code units, meaning the length property is 3 rather than 2. The charCodeAt() method returns only the first code unit for position 0, but codePointAt() returns the full code point even though the code point spans multiple code units. Both methods return the same value for positions 1 (the second code unit of the first character) and 2 (the "a" character).

Calling the codePointAt() method on a character is the easiest way to determine if that character is represented by one or two code points. Here's a function you could write to check:

```
function is32Bit(c) {
    return c.codePointAt(0) > 0xFFFF;
}

console.log(is32Bit(" "));          // true
console.log(is32Bit("a"));          // false
```

The upper bound of 16-bit characters is represented in hexadecimal as `FFFF`, so any code point above that number must be represented by two code units, for a total of 32 bits.

The String.fromCodePoint() Method

When ECMAScript provides a way to do something, it also tends to provide a way to do the reverse. You can use `codePointAt()` to retrieve the code point for a character in a string, while `String.fromCodePoint()` produces a single-character string from a given code point. For example:

```
console.log(String.fromCodePoint(134071));  // " "
```

Think of `String.fromCodePoint()` as a more complete version of the `String.fromCharCode()` method. Both give the same result for all characters in the BMP. There's only a difference when you pass code points for characters outside of the BMP.

The normalize() Method

Another interesting aspect of Unicode is that different characters may be considered equivalent for the purpose of sorting or other comparison-based operations. There are two ways to define these relationships. First, canonical equivalence means that two sequences of code points are considered interchangeable in all respects. For example, a combination of two characters can be canonically equivalent to one character. The second relationship is compatibility. Two compatible sequences of code points look different but can be used interchangeably in certain situations.

Due to these relationships, two strings representing fundamentally the same text can contain different code point sequences. For example, the character “æ” and the two-character string “ae” may be used interchangeably but are strictly not equivalent unless normalized in some way.

ECMAScript 6 supports Unicode normalization forms by giving strings a `normalize()` method. This method optionally accepts a single string parameter indicating one of the following Unicode normalization forms to apply:

- Normalization Form Canonical Composition ("NFC"), which is the default
- Normalization Form Canonical Decomposition ("NFD")
- Normalization Form Compatibility Composition ("NFKC")
- Normalization Form Compatibility Decomposition ("NFKD")

It's beyond the scope of this book to explain the differences between these four forms. Just keep in mind that when comparing strings, both strings must be normalized to the same form. For example:

```
var normalized = values.map(function(text) {
    return text.normalize();
});

normalized.sort(function(first, second) {
    if (first < second) {
        return -1;
    } else if (first === second) {
        return 0;
    } else {
        return 1;
    }
});
```

This code converts the strings in the `values` array into a normalized form so that the array can be sorted appropriately. You can also sort the original array by calling `normalize()` as part of the comparator, as follows:

```
values.sort(function(first, second) {
    var firstNormalized = first.normalize(),
        secondNormalized = second.normalize();

    if (firstNormalized < secondNormalized) {
        return -1;
    } else if (firstNormalized === secondNormalized) {
        return 0;
    } else {
        return 1;
    }
});
```

Once again, the most important thing to note about this code is that both `first` and `second` are normalized in the same way. These examples have used the default, NFC, but you can just as easily specify one of the others, like this:

```
values.sort(function(first, second) {
    var firstNormalized = first.normalize("NFD"),
        secondNormalized = second.normalize("NFD");

    if (firstNormalized < secondNormalized) {
        return -1;
    } else if (firstNormalized === secondNormalized) {
        return 0;
    } else {
        return 1;
    }
});
```

If you've never worried about Unicode normalization before, then you probably won't have much use for this method now. But if you ever work on an internationalized application, you'll definitely find the normalize() method helpful.

Methods aren't the only improvements that ECMAScript 6 provides for working with Unicode strings, though. The standard also adds two useful syntax elements.

The Regular Expression u Flag

You can accomplish many common string operations through regular expressions. But remember, regular expressions assume 16-bit code units, where each represents a single character. To address this problem, ECMAScript 6 defines a u flag for regular expressions, which stands for Unicode.

The u Flag in Action

When a regular expression has the u flag set, it switches modes to work on characters, not code units. That means the regular expression should no longer get confused about surrogate pairs in strings and should behave as expected. For example, consider this code:

```
var text = " ";

console.log(text.length);           // 2
console.log(/^.$/.test(text));      // false
console.log(/^.$/u.test(text));     // true
```

The regular expression /^.$/ matches any input string with a single character.

When used without the u flag, this regular expression matches on code units, and so the Japanese character (which is represented by two code units) doesn't match the regular expression. When used with the u flag, the regular expression compares characters instead of code units and so the Japanese character matches.

Counting Code Points

Unfortunately, ECMAScript 6 can't natively determine how many code points a string has, but with the u flag, you can use regular expressions to figure it out as follows:

```
function codePointLength(text) {
    var result = text.match(/[\s\S]/gu);
    return result ? result.length : 0;
}

console.log(codePointLength("abc"));    // 3
console.log(codePointLength(" bc"));   // 3
```

This example calls match() to check text for both whitespace and non-whitespace characters (using [\s\S] to ensure the pattern matches newlines), using a regular expression that is applied globally with Unicode enabled. The result contains an array of matches when there's at least one match, so the array length is the number of code points in the string. In Unicode, the strings "abc" and " bc" both have three characters, so the array length is three.

⚠ Although this approach works, it's not very fast, especially when applied to long strings. You can use a string iterator (discussed in Chapter 8) as well. In general, try to minimize counting code points whenever possible.

Determining Support for the u Flag

Since the u flag is a syntax change, attempting to use it in JavaScript engines that aren't compatible with ECMAScript 6 throws a syntax error. The safest way to determine if the u flag is supported is with a function, like this one:

```
function hasRegExpU() {
    try {
        var pattern = new RegExp(".", "u");
        return true;
    } catch (ex) {
        return false;
    }
}
```

This function uses the RegExp constructor to pass in the u flag as an argument. This syntax is valid even in older JavaScript engines, but the constructor will throw an error if u isn't supported.

If your code still needs to work in older JavaScript engines, always use the RegExp constructor when using the u flag. This will prevent syntax errors and allow you to optionally detect and use the u flag without aborting execution.

Other String Changes

JavaScript strings have always lagged behind similar features of other languages. It was only in ECMAScript 5 that strings finally gained a trim() method, for example, and ECMAScript 6 continues extending JavaScript's capacity to parse strings with new functionality.

Methods for Identifying Substrings

Developers have used the indexOf() method to identify strings inside other strings since JavaScript was first introduced. ECMAScript 6 includes the following three methods, which are designed to do just that:

- The includes() method returns true if the given text is found anywhere within the string. It returns false if not.
- The startsWith() method returns true if the given text is found at the beginning of the string. It returns false if not.
- The endsWith() method returns true if the given text is found at the end of the string. It returns false if not.

Each methods accept two arguments: the text to search for and an optional index from which to start the search. When the second argument is provided, includes() and startsWith() start the match from that index while endsWith() starts the match from the length of the string minus the second argument; when the second argument is omitted, includes() and startsWith() search from the beginning of the string, while endsWith() starts from the end. In effect, the second argument minimizes the amount of the string being searched. Here are some examples showing these three methods in action:

```
var msg = "Hello world!";

console.log(msg.startsWith("Hello"));       // true
console.log(msg.endsWith("!"));             // true
console.log(msg.includes("o"));             // true

console.log(msg.startsWith("o"));           // false
console.log(msg.endsWith("world!"));        // true
console.log(msg.includes("x"));             // false

console.log(msg.startsWith("o", 4));        // true
console.log(msg.endsWith("o", 8));          // true
console.log(msg.includes("o", 8));          // false
```

The first three calls don't include a second parameter, so they'll search the whole string if needed. The last three calls only check part of the string. The call to msg.startsWith("o", 4) starts the match by looking at index 4 of the msg string, which is the "o" in "Hello". The call to msg.endsWith("o", 8) starts the match at index 4 as well, because the 8 argument is subtracted from the string length (12). The call to msg.includes("o", 8) starts the match from index 8, which is the "r" in "world".

While these three methods make identifying the existence of substrings easier, each only returns a boolean value. If you need to find the actual position of one string within another, use the indexOf() or lastIndexOf() methods.

⚠ The startsWith(), endsWith(), and includes() methods will throw an error if you pass a regular expression instead of a string. This stands in contrast to indexOf() and lastIndexOf(), which both convert a regular expression argument into a string and then search for that string.

The repeat() Method

ECMAScript 6 also adds a `repeat()` method to strings, which accepts the number of times to repeat the string as an argument. It returns a new string containing the original string repeated the specified number of times. For example:

```
console.log("x".repeat(3));          // "xxx"
console.log("hello".repeat(2));      // "hellohello"
console.log("abc".repeat(4));        // "abcabcabcabc"
```

This method is a convenience function above all else, and it can be especially useful when manipulating text. It’s particularly useful in code formatting utilities that need to create indentation levels, like this:

```
// indent using a specified number of spaces
var indent = " ".repeat(4),
    indentLevel = 0;

// whenever you increase the indent
var newIndent = indent.repeat(++indentLevel);
```

The first `repeat()` call creates a string of four spaces, and the `indentLevel` variable keeps track of the indent level. Then, you can just call `repeat()` with an incremented `indentLevel` to change the number of spaces.

ECMAScript 6 also makes some useful changes to regular expression functionality that don’t fit into a particular category. The next section highlights a few.

Other Regular Expression Changes

Regular expressions are an important part of working with strings in JavaScript, and like many parts of the language, they haven’t changed much in recent versions. ECMAScript 6, however, makes several improvements to regular expressions to go along with the updates to strings.

The Regular Expression y Flag

ECMAScript 6 standardized the `y` flag after it was implemented in Firefox as a proprietary extension to regular expressions. The `y` flag affects a regular expression search’s `sticky` property, and it tells the search to start matching characters in a string at the position specified by the regular expression’s `lastIndex` property. If there is no match at that location, then the regular expression stops matching. To see how this works, consider the following code:

```
var text = "hello1 hello2 hello3",
    pattern = /hello\d\s?/,
    result = pattern.exec(text),
    globalPattern = /hello\d\s?/g,
    globalResult = globalPattern.exec(text),
    stickyPattern = /hello\d\s?/y,
    stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello1 "
console.log(stickyResult[0]);   // "hello1 "

pattern.lastIndex = 1;
globalPattern.lastIndex = 1;
stickyPattern.lastIndex = 1;

result = pattern.exec(text);
globalResult = globalPattern.exec(text);
stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello2 "
console.log(stickyResult[0]);   // Error! stickyResult is null
```

This example has three regular expressions. The expression in pattern has no flags, the one in globalPattern uses the g flag, and the one in stickyPattern uses the y flag. In the first trio of console.log() calls, all three regular expressions should return "hello1 " with a space at the end.

After that, the lastIndex property is changed to 1 on all three patterns, meaning that the regular expression should start matching from the second character on all of them. The regular expression with no flags completely ignores the change to lastIndex and still matches "hello1 " without incident. The regular expression with the g flag goes on to match "hello2 " because it is searching forward from the second character of the string ("e"). The sticky regular expression doesn't match anything beginning at the second character so stickyResult is null.

The sticky flag saves the index of the next character after the last match in lastIndex whenever an operation is performed. If an operation results in no match, then lastIndex is set back to 0. The global flag behaves the same way, as demonstrated here:

```
var text = "hello1 hello2 hello3",
    pattern = /hello\d\s?/,
    result = pattern.exec(text),
    globalPattern = /hello\d\s?/g,
    globalResult = globalPattern.exec(text),
    stickyPattern = /hello\d\s?/y,
    stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello1 "
console.log(stickyResult[0]);   // "hello1 "

console.log(pattern.lastIndex);         // 0
console.log(globalPattern.lastIndex);   // 7
console.log(stickyPattern.lastIndex);   // 7

result = pattern.exec(text);
globalResult = globalPattern.exec(text);
stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello2 "
console.log(stickyResult[0]);   // "hello2 "

console.log(pattern.lastIndex);         // 0
console.log(globalPattern.lastIndex);   // 14
console.log(stickyPattern.lastIndex);   // 14
```

The value of `lastIndex` changes to 7 after the first call to `exec()` and to 14 after the second call, for both the `stickyPattern` and `globalPattern` variables.

There are two more subtle details about the sticky flag to keep in mind:

1. The `lastIndex` property is only honored when calling methods that exist on the regular expression object, like the `exec()` and `test()` methods. Passing the regular expression to a string method, such as `match()`, will not result in the sticky behavior.
2. When using the `^` character to match the start of a string, sticky regular expressions only match from the start of the string (or the start of the line in multiline mode). While `lastIndex` is 0, the `^` makes a sticky regular expression no

different from a non-sticky one. If lastIndex doesn't correspond to the beginning of the string in single-line mode or the beginning of a line in multiline mode, the sticky regular expression will never match.

As with other regular expression flags, you can detect the presence of y by using a property. In this case, you'd check the sticky property, as follows:

```
var pattern = /hello\d/y;

console.log(pattern.sticky);    // true
```

The sticky property is set to true if the sticky flag is present, and the property is false if not. The sticky property is read-only based on the presence of the flag and cannot be changed in code.

Similar to the u flag, the y flag is a syntax change, so it will cause a syntax error in older JavaScript engines. You can use the following approach to detect support:

```
function hasRegExpY() {
    try {
        var pattern = new RegExp(".", "y");
        return true;
    } catch (ex) {
        return false;
    }
}
```

Just like the u check, this returns false if it's unable to create a regular expression with the y flag. In one final similarity to u, if you need to use y in code that runs in older JavaScript engines, be sure to use the RegExp constructor when defining those regular expressions to avoid a syntax error.

Duplicating Regular Expressions

In ECMAScript 5, you can duplicate regular expressions by passing them into the RegExp constructor like this:

```
var re1 = /ab/i,
    re2 = new RegExp(re1);
```

The re2 variable is just a copy of the re1 variable. But if you provide the second argument to the RegExp constructor, which specifies the flags for the regular expression, your code won't work, as in this example:

```
var re1 = /ab/i,

    // throws an error in ES5, okay in ES6
    re2 = new RegExp(re1, "g");
```

If you execute this code in an ECMAScript 5 environment, you'll get an error stating that the second argument cannot be used when the first argument is a regular expression. ECMAScript 6 changed this behavior such that the second argument is allowed and overrides any flags present on the first argument. For example:

```
var re1 = /ab/i,

    // throws an error in ES5, okay in ES6
    re2 = new RegExp(re1, "g");


console.log(re1.toString());          // "/ab/i"
console.log(re2.toString());          // "/ab/g"

console.log(re1.test("ab"));          // true
console.log(re2.test("ab"));          // true

console.log(re1.test("AB"));           // true
console.log(re2.test("AB"));           // false
```

In this code, re1 has the case-insensitive i flag while re2 has only the global g flag. The RegExp constructor duplicated the pattern from re1 and substituted the g flag for the i flag. Without the second argument, re2 would have the same flags as re1.

The flags Property

Along with adding a new flag and changing how you can work with flags, ECMAScript 6 added a property associated with them. In ECMAScript 5, you could get the text of a regular expression by using the source property, but to get the flag string, you'd have to parse the output of the toString() method as shown below:

```
function getFlags(re) {
    var text = re.toString();
    return text.substring(text.lastIndexOf("/") + 1, text.length);
}

// toString() is "/ab/g"
var re = /ab/g;

console.log(getFlags(re));          // "g"
```

This converts a regular expression into a string and then returns the characters found after the last `/`. Those characters are the flags.

ECMAScript 6 makes fetching flags easier by adding a `flags` property to go along with the `source` property. Both properties are prototype accessor properties with only a getter assigned, making them read-only. The `flags` property makes inspecting regular expressions easier for both debugging and inheritance purposes.

A late addition to ECMAScript 6, the `flags` property returns the string representation of any flags applied to a regular expression. For example:

```
var re = /ab/g;

console.log(re.source);     // "ab"
console.log(re.flags);      // "g"
```

This fetches all flags on `re` and prints them to the console with far fewer lines of code than the `toString()` technique can. Using `source` and `flags` together allows you to extract the pieces of the regular expression that you need without parsing the regular expression string directly.

The changes to strings and regular expressions that this chapter has covered so far are definitely powerful, but ECMAScript 6 improves your power over strings in a much bigger way. It brings a type of literal to the table that makes strings more flexible.

Template Literals

JavaScript's strings have always had limited functionality compared to strings in other languages. For instance, until ECMAScript 6, strings lacked the methods covered so far in this chapter, and string concatenation is as simple as possible. To allow developers to solve more complex problems, ECMAScript 6's template literals provide syntax for creating domain-specific languages (DSLs) for working with content in a safer way than the solutions available in ECMAScript 5 and earlier. (A DSL is a programming language designed for a specific, narrow purpose, as opposed to general-purpose languages like JavaScript.) The ECMAScript wiki offers the following description on the template literal strawman:

> This scheme extends ECMAScript syntax with syntactic sugar to allow libraries to provide DSLs that easily produce, query, and manipulate content from other languages that are immune or resistant to injection attacks such as XSS, SQL Injection, etc.

In reality, though, template literals are ECMAScript 6's answer to the following features that JavaScript lacked all the way through ECMAScript 5:

- **Multiline strings** A formal concept of multiline strings.
- **Basic string formatting** The ability to substitute parts of the string for values contained in variables.
- **HTML escaping** The ability to transform a string such that it is safe to insert into HTML.

Rather than trying to add more functionality to JavaScript's already-existing strings, template literals represent an entirely new approach to solving these problems.

Basic Syntax

At their simplest, template literals act like regular strings delimited by backticks (`) instead of double or single quotes. For example, consider the following:

```
let message = `Hello world!`;

console.log(message);               // "Hello world!"
console.log(typeof message);        // "string"
console.log(message.length);        // 12
```

This code demonstrates that the variable `message` contains a normal JavaScript string. The template literal syntax is used to create the string value, which is then assigned to the `message` variable.

If you want to use a backtick in your string, then just escape it with a backslash (\), as in this version of the `message` variable:

```
let message = `\`Hello\` world!`;

console.log(message);             // "`Hello` world!"
console.log(typeof message);       // "string"
console.log(message.length);       // 14
```

There’s no need to escape either double or single quotes inside of template literals.

Multiline Strings

JavaScript developers have wanted a way to create multiline strings since the first version of the language. But when using double or single quotes, strings must be completely contained on a single line.

Pre-ECMAScript 6 Workarounds

Thanks to a long-standing syntax bug, JavaScript does have a workaround. You can create multiline strings if there’s a backslash (\) before a newline. Here’s an example:

```
var message = "Multiline \
string";

console.log(message);       // "Multiline string"
```

The `message` string has no newlines present when printed to the console because the backslash is treated as a continuation rather than a newline. In order to show a newline in output, you’d need to manually include it:

```
var message = "Multiline \n\
string";

console.log(message);       // "Multiline
                            //  string"
```

This should print `Multiline String` on two separate lines in all major JavaScript engines, but the behavior is defined as a bug and many developers recommend avoiding it.

Other pre-ECMAScript 6 attempts to create multiline strings usually relied on arrays or string concatenation, such as:

```
var message = [
    "Multiline ",
    "string"
].join("\n");

let message = "Multiline \n" +
    "string";
```

All of the ways developers worked around JavaScript’s lack of multiline strings left something to be desired.

Multiline Strings the Easy Way

ECMAScript 6’s template literals make multiline strings easy because there’s no special syntax. Just include a newline where you want, and it shows up in the result. For example:

```
let message = `Multiline
string`;

console.log(message);           // "Multiline
                                //  string"
console.log(message.length);    // 16
```

All whitespace inside the backticks is part of the string, so be careful with indentation. For example:

```
let message = `Multiline
               string`;

console.log(message);           // "Multiline
                                //                string"
console.log(message.length);    // 31
```

In this code, all whitespace before the second line of the template literal is considered part of the string itself. If making the text line up with proper indentation is important to you, then consider leaving nothing on the first line of a multiline template literal and then indenting after that, as follows:

```
let html = `
<div>
    <h1>Title</h1>
</div>`.trim();
```

This code begins the template literal on the first line but doesn't have any text until the second. The HTML tags are indented to look correct and then the `trim()` method is called to remove the initial empty line.

> If you prefer, you can also use `\n` in a template literal to indicate where a newline should be inserted:
>
> ```
> let message = `Multiline\nstring`;
>
> console.log(message);         // "Multiline
>                               //  string"
> console.log(message.length);   // 16
> ```

Making Substitutions

At this point, template literals may look like fancier versions of normal JavaScript strings. The real difference between the two lies in template literal substitutions. Substitutions allow you to embed any valid JavaScript expression inside a template literal and output the result as part of the string.

Substitutions are delimited by an opening `${` and a closing `}` that can have any JavaScript expression inside. The simplest substitutions let you embed local variables directly into a resulting string, like this:

```
let name = "Nicholas",
    message = `Hello, ${name}.`;

console.log(message);       // "Hello, Nicholas."
```

The substitution `${name}` accesses the local variable `name` to insert `name` into the `message` string. The `message` variable then holds the result of the substitution immediately.

A template literal can access any variable accessible in the scope in which it is defined. Attempting to use an undeclared variable in a template literal throws an error in both strict and non-strict modes.

Since all substitutions are JavaScript expressions, you can substitute more than just simple variable names. You can easily embed calculations, function calls, and more. For example:

```
let count = 10,
    price = 0.25,
    message = `${count} items cost $${(count * price).toFixed(2)}.`;

console.log(message);       // "10 items cost $2.50."
```

This code performs a calculation as part of the template literal. The variables `count` and `price` are multiplied together to get a result, and then formatted to two decimal places using `.toFixed()`. The dollar sign before the second substitution is output as-is because it's not followed by an opening curly brace.

Template literals are also JavaScript expressions, which means you can place a template literal inside of another template literal, as in this example:

```
let name = "Nicholas",
    message = `Hello, ${
        `my name is ${ name }`
    }.`;

console.log(message);        // "Hello, my name is Nicholas."
```

This example nests a second template literal inside the first. After the first `${`, another template literal begins. The second `${` indicates the beginning of an embedded expression inside the inner template literal. That expression is the variable `name`, which is inserted into the result.

Tagged Templates

Now you've seen how template literals can create multiline strings and insert values into strings without concatenation. But the real power of template literals comes from tagged templates. A template tag performs a transformation on the template literal and returns the final string value. This tag is specified at the start of the template, just before the first ` character, as shown here:

```
let message = tag`Hello world`;
```

In this example, `tag` is the template tag to apply to the `` `Hello world` `` template literal.

Defining Tags

A tag is simply a function that is called with the processed template literal data. The tag receives data about the template literal as individual pieces and must combine the pieces to create the result. The first argument is an array containing the literal strings as interpreted by JavaScript. Each subsequent argument is the interpreted value of each substitution.

Tag functions are typically defined using rest arguments as follows, to make dealing with the data easier:

```
function tag(literals, ...substitutions) {
    // return a string
}
```

To better understand what gets passed to tags, consider the following:

```
let count = 10,
    price = 0.25,
    message = passthru`${count} items cost $${(count * price).toFixed(2)}.`;
```

If you had a function called `passthru()`, that function would receive three arguments. First, it would get a `literals` array, containing the following elements:

- The empty string before the first substitution ("")
- The string after the first substitution and before the second (" items cost $")
- The string after the second substitution (".")

The next argument would be `10`, which is the interpreted value for the `count` variable. This becomes the first element in a `substitutions` array. The final argument would be `"2.50"`, which is the interpreted value for `(count * price).toFixed(2)` and the second element in the `substitutions` array.

Note that the first item in `literals` is an empty string. This ensures that `literals[0]` is always the start of the string, just like `literals[literals.length - 1]` is always the end of the string. There is always one fewer substitution than literal, which means the expression `substitutions.length === literals.length - 1` is always true.

Using this pattern, the `literals` and `substitutions` arrays can be interwoven to create a resulting string. The first item in `literals` comes first, the first item in `substitutions` is next, and so on, until the string is complete. As an example, you can mimic the default behavior of a template literal by alternating values from these two arrays:

```
function passthru(literals, ...substitutions) {
    let result = "";

    // run the loop only for the substitution count
    for (let i = 0; i < substitutions.length; i++) {
        result += literals[i];
        result += substitutions[i];
    }

    // add the last literal
    result += literals[literals.length - 1];

    return result;
}

let count = 10,
    price = 0.25,
    message = passthru`${count} items cost $${(count * price).toFixed(2)}.`;

console.log(message);       // "10 items cost $2.50."
```

This example defines a passthru tag that performs the same transformation as the default template literal behavior. The only trick is to use substitutions.length for the loop rather than literals.length to avoid accidentally going past the end of the substitutions array. This works because the relationship between literals and substitutions is well-defined in ECMAScript 6.

The values contained in substitutions are not necessarily strings. If an expression evaluates to a number, as in the previous example, then the numeric value is passed in. Determining how such values should output in the result is part of the tag's job.

Using Raw Values in Template Literals

Template tags also have access to raw string information, which primarily means access to character escapes before they are transformed into their character equivalents. The simplest way to work with raw string values is to use the built-in String.raw() tag. For example:

```
let message1 = `Multiline\nstring`,
    message2 = String.raw`Multiline\nstring`;

console.log(message1);          // "Multiline
                                //  string"
console.log(message2);          // "Multiline\\nstring"
```

In this code, the \n in message1 is interpreted as a newline while the \n in message2 is returned in its raw form of "\\n" (the slash and n characters). Retrieving the raw string information like this allows for more complex processing when necessary.

The raw string information is also passed into template tags. The first argument in a tag function is an array with an extra property called raw. The raw property is an array containing the raw equivalent of each literal value. For example, the value in literals[0] always has an equivalent literals.raw[0] that contains the raw string information. Knowing that, you can mimic String.raw() using the following code:

```
function raw(literals, ...substitutions) {
    let result = "";

    // run the loop only for the substitution count
    for (let i = 0; i < substitutions.length; i++) {
        result += literals.raw[i];      // use raw values instead
        result += substitutions[i];
    }

    // add the last literal
    result += literals.raw[literals.length - 1];

    return result;
}

let message = raw`Multiline\nstring`;

console.log(message);           // "Multiline\\nstring"
console.log(message.length);    // 17
```

This uses literals.raw instead of literals to output the string result. That means any character escapes, including Unicode code point escapes, should be returned in their raw form.Raw strings are helpful when you want to output a string containing code in which you'll need to include the character escaping (for instance, if you want to generate documentation about some code, you may want to output the actual code as it appears).

Summary

Full Unicode support allows JavaScript to deal with UTF-16 characters in logical ways. The ability to transfer between code point and character via `codePointAt()` and `String.fromCodePoint()` is an important step for string manipulation. The addition of the regular expression `u` flag makes it possible to operate on code points instead of 16-bit characters, and the `normalize()` method allows for more appropriate string comparisons.

ECMAScript 6 also added new methods for working with strings, allowing you to more easily identify a substring regardless of its position in the parent string. More functionality was added to regular expressions, too.

Template literals are an important addition to ECMAScript 6 that allows you to create domain-specific languages (DSLs) to make creating strings easier. The ability to embed variables directly into template literals means that developers have a safer tool than string concatenation for composing long strings with variables.

Built-in support for multiline strings also makes template literals a useful upgrade over normal JavaScript strings, which have never had this ability. Despite allowing newlines directly inside the template literal, you can still use `\n` and other character escape sequences.

Template tags are the most important part of this feature for creating DSLs. Tags are functions that receive the pieces of the template literal as arguments. You can then use that data to return an appropriate string value. The data provided includes literals, their raw equivalents, and any substitution values. These pieces of information can then be used to determine the correct output for the tag.

Functions

Functions are an important part of any programming language, and prior to ECMAScript 6, JavaScript functions hadn’t changed much since the language was created. This left a backlog of problems and nuanced behavior that made making mistakes easy and often required more code just to achieve very basic behaviors.

ECMAScript 6 functions make a big leap forward, taking into account years of complaints and requests from JavaScript developers. The result is a number of incremental improvements on top of ECMAScript 5 functions that make programming in JavaScript less error-prone and more powerful.

Functions with Default Parameter Values

Functions in JavaScript are unique in that they allow any number of parameters to be passed, regardless of the number of parameters declared in the function definition. This allows you to define functions that can handle different numbers of parameters, often by just filling in default values when parameters aren't provided. This section covers how default parameters work both in and prior to ECMAScript 6, along with some important information on the arguments object, using expressions as parameters, and another TDZ.

Simulating Default Parameter Values in ECMAScript 5

In ECMAScript 5 and earlier, you would likely use the following pattern to create a function with default parameters values:

```
function makeRequest(url, timeout, callback) {

    timeout = timeout || 2000;
    callback = callback || function() {};

    // the rest of the function

}
```

In this example, both timeout and callback are actually optional because they are given a default value if a parameter isn't provided. The logical OR operator (||) always returns the second operand when the first is falsy. Since named function parameters that are not explicitly provided are set to undefined, the logical OR operator is frequently used to provide default values for missing parameters. There is a flaw with this approach, however, in that a valid value for timeout might actually be 0, but this would replace it with 2000 because 0 is falsy.

In that case, a safer alternative is to check the type of the argument using typeof, as in this example:

```
function makeRequest(url, timeout, callback) {

    timeout = (typeof timeout !== "undefined") ? timeout : 2000;
    callback = (typeof callback !== "undefined") ? callback : function() {};

    // the rest of the function

}
```

While this approach is safer, it still requires a lot of extra code for a very basic operation. Popular JavaScript libraries are filled with similar patterns, as this represents a common pattern.

Default Parameter Values in ECMAScript 6

ECMAScript 6 makes it easier to provide default values for parameters by providing initializations that are used when the parameter isn't formally passed. For example:

```
function makeRequest(url, timeout = 2000, callback = function() {}) {

    // the rest of the function

}
```

This function only expects the first parameter to always be passed. The other two parameters have default values, which makes the body of the function much smaller because you don't need to add any code to check for a missing value.

When `makeRequest()` is called with all three parameters, the defaults are not used. For example:

```
// uses default timeout and callback
makeRequest("/foo");

// uses default callback
makeRequest("/foo", 500);

// doesn't use defaults
makeRequest("/foo", 500, function(body) {
    doSomething(body);
});
```

ECMAScript 6 considers `url` to be required, which is why `"/foo"` is passed in all three calls to `makeRequest()`. The two parameters with a default value are considered optional.

It's possible to specify default values for any arguments, including those that appear before arguments without default values in the function declaration. For example, this is fine:

```
function makeRequest(url, timeout = 2000, callback) {

    // the rest of the function

}
```

In this case, the default value for timeout will only be used if there is no second argument passed in or if the second argument is explicitly passed in as undefined, as in this example:

```
// uses default timeout
makeRequest("/foo", undefined, function(body) {
    doSomething(body);
});

// uses default timeout
makeRequest("/foo");

// doesn't use default timeout
makeRequest("/foo", null, function(body) {
    doSomething(body);
});
```

In the case of default parameter values, a value of null is considered to be valid, meaning that in the third call to makeRequest(), the default value for timeout will not be used.

How Default Parameter Values Affect the arguments Object

Just keep in mind that the behavior of the arguments object is different when default parameter values are present. In ECMAScript 5 nonstrict mode, the arguments object reflects changes in the named parameters of a function. Here’s some code that illustrates how this works:

```
function mixArgs(first, second) {
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d";
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}

mixArgs("a", "b");
```

This outputs:

```
true
true
true
true
```

The `arguments` object is always updated in nonstrict mode to reflect changes in the named parameters. Thus, when `first` and `second` are assigned new values, `arguments[0]` and `arguments[1]` are updated accordingly, making all of the `===` comparisons resolve to `true`.

ECMAScript 5's strict mode, however, eliminates this confusing aspect of the `arguments` object. In strict mode, the `arguments` object does not reflect changes to the named parameters. Here's the `mixArgs()` function again, but in strict mode:

```
function mixArgs(first, second) {
    "use strict";

    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d"
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}

mixArgs("a", "b");
```

The call to `mixArgs()` outputs:

```
true
true
false
false
```

This time, changing first and second doesn't affect arguments, so the output behaves as you'd normally expect it to.

The arguments object in a function using ECMAScript 6 default parameter values, however, will always behave in the same manner as ECMAScript 5 strict mode, regardless of whether the function is explicitly running in strict mode. The presence of default parameter values triggers the arguments object to remain detached from the named parameters. This is a subtle but important detail because of how the arguments object may be used. Consider the following:

```
// not in strict mode
function mixArgs(first, second = "b") {
    console.log(arguments.length);
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d"
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}

mixArgs("a");
```

This outputs:

```
1
true
false
false
false
```

In this example, arguments.length is 1 because only one argument was passed to mixArgs(). That also means arguments[1] is undefined, which is the expected behavior when only one argument is passed to a function. That means first is equal to arguments[0] as well. Changing first and second has no effect on arguments. This behavior occurs in both nonstrict and strict mode, so you can rely on arguments to always reflect the initial call state.

Default Parameter Expressions

Perhaps the most interesting feature of default parameter values is that the default value need not be a primitive value. You can, for example, execute a function to retrieve the default parameter value, like this:

```
function getValue() {
    return 5;
}

function add(first, second = getValue()) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 6
```

Here, if the last argument isn't provided, the function getValue() is called to retrieve the correct default value. Keep in mind that getValue() is only called when add() is called without a second parameter, not when the function declaration is first parsed. That means if getValue() were written differently, it could potentially return a different value. For instance:

```
let value = 5;

function getValue() {
    return value++;
}

function add(first, second = getValue()) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 6
console.log(add(1));        // 7
```

In this example, value begins as five and increments each time getValue() is called. The first call to add(1) returns 6, while the second call to add(1) returns 7 because value was incremented. Because the default value for second is only evaluated when the function is called, changes to that value can be made at any time.

Be careful when using function calls as default parameter values. If you forget the parentheses, such as `second = getValue` in the last example, you are passing a reference to the function rather than the result of the function call.

This behavior introduces another interesting capability. You can use a previous parameter as the default for a later parameter. Here's an example:

```
function add(first, second = first) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 2
```

In this code, the parameter `second` is given a default value of `first`, meaning that passing in just one argument leaves both arguments with the same value. So `add(1, 1)` returns 2 just as `add(1)` returns 2. Taking this a step further, you can pass `first` into a function to get the value for `second` as follows:

```
function getValue(value) {
    return value + 5;
}

function add(first, second = getValue(first)) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 7
```

This example sets `second` equal to the value returned by `getValue(first)`, so while `add(1, 1)` still returns 2, `add(1)` returns 7 (1 + 6).

The ability to reference parameters from default parameter assignments works only for previous arguments, so earlier arguments do not have access to later arguments. For example:

```
function add(first = second, second) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // throws error
```

The call to add(1) throws an error because second is defined after first and is therefore unavailable as a default value. To understand why that happens, it's important to revisit temporal dead zones.

Default Parameter Value Temporal Dead Zone

Chapter 1 introduced the temporal dead zone (TDZ) as it relates to let and const, and default parameter values also have a TDZ where parameters cannot be accessed. Similar to a let declaration, each parameter creates a new identifier binding that can't be referenced before initialization without throwing an error. Parameter initialization happens when the function is called, either by passing a value for the parameter or by using the default parameter value.

To explore the default parameter value TDZ, consider this example from "Default Parameter Expressions" again:

```
function getValue(value) {
    return value + 5;
}

function add(first, second = getValue(first)) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 7
```

The calls to add(1, 1) and add(1) effectively execute the following code to create the first and second parameter values:

```
// JavaScript representation of call to add(1, 1)
let first = 1;
let second = 1;

// JavaScript representation of call to add(1)
let first = 1;
let second = getValue(first);
```

When the function add() is first executed, the bindings first and second are added to a parameter-specific TDZ (similar to how let behaves). So while second can be initialized with the value of first because first is always initialized at that time, the reverse is not true. Now, consider this rewritten add() function:

```
function add(first = second, second) {
    return first + second;
}

console.log(add(1, 1));         // 2
console.log(add(undefined, 1)); // throws error
```

The calls to add(1, 1) and add(undefined, 1) in this example now map to this code behind the scenes:

```
// JavaScript representation of call to add(1, 1)
let first = 1;
let second = 1;

// JavaScript representation of call to add(undefined, 1)
let first = second;
let second = 1;
```

In this example, the call to add(undefined, 1) throws an error because second hasn't yet been initialized when first is initialized. At that point, second is in the TDZ and therefore any references to second throw an error. This mirrors the behavior of let bindings discussed in Chapter 1.

Function parameters have their own scope and their own TDZ that is separate from the function body scope. That means the default value of a parameter cannot access any variables declared inside the function body.

Working with Unnamed Parameters

So far, the examples in this chapter have only covered parameters that have been named in the function definition. However, JavaScript functions don't limit the number of parameters that can be passed to the number of named parameters defined. You can always pass fewer or more parameters than formally specified. Default parameter values make it clear when a function can accept fewer parameters, and ECMAScript 6 sought to make the problem of passing more parameters than defined better as well.

Unnamed Parameters in ECMAScript 5

Early on, JavaScript provided the `arguments` object as a way to inspect all function parameters that are passed without necessarily defining each parameter individually. While inspecting `arguments` works fine in most cases, this object can be a little cumbersome to work with. For example, examine this code, which inspects the `arguments` object:

```
function pick(object) {
    let result = Object.create(null);

    // start at the second parameter
    for (let i = 1, len = arguments.length; i < len; i++) {
        result[arguments[i]] = object[arguments[i]];
    }

    return result;
}

let book = {
    title: "Understanding ECMAScript 6",
    author: "Nicholas C. Zakas",
    year: 2015
};

let bookData = pick(book, "author", "year");

console.log(bookData.author);   // "Nicholas C. Zakas"
console.log(bookData.year);     // 2015
```

This function mimics the `pick()` method from the Underscore.js library, which returns a copy of a given object with some specified subset of the original object’s properties. This example defines only one argument and expects the first argument to be the object from which to copy properties. Every other argument passed is the name of a property that should be copied on the result.

There are couple of things to notice about this `pick()` function. First, it’s not at all obvious that the function can handle more than one parameter. You could define several more parameters, but you would always fall short of indicating that this function can take any number of parameters. Second, because the first parameter is named and used directly, when you look for the properties to copy, you have to start in the `arguments` object at index 1 instead of index 0. Remembering to use the appropriate indices with `arguments` isn’t necessarily difficult, but it’s one more thing to keep track of.

ECMAScript 6 introduces rest parameters to help with these issues.

Rest Parameters

A rest parameter is indicated by three dots (...) preceding a named parameter. That named parameter becomes an `Array` containing the rest of the parameters passed to the function, which is where the name “rest” parameters originates. For example, pick() can be rewritten using rest parameters, like this:

```
function pick(object, ...keys) {
    let result = Object.create(null);

    for (let i = 0, len = keys.length; i < len; i++) {
        result[keys[i]] = object[keys[i]];
    }

    return result;
}
```

In this version of the function, keys is a rest parameter that contains all parameters passed after object (unlike arguments, which contains all parameters including the first one). That means you can iterate over keys from beginning to end without worry. As a bonus, you can tell by looking at the function that it is capable of handling any number of parameters.

Rest parameters do not affect a function’s length property, which indicates the number of named parameters for the function. The value of length for pick() in this example is 1 because only object counts towards this value.

Rest Parameter Restrictions

There are two restrictions on rest parameters. The first restriction is that there can be only one rest parameter, and the rest parameter must be last. For example, this code won’t work:

```
// Syntax error: Can't have a named parameter after rest parameters
function pick(object, ...keys, last) {
    let result = Object.create(null);

    for (let i = 0, len = keys.length; i < len; i++) {
        result[keys[i]] = object[keys[i]];
    }

    return result;
}
```

Here, the parameter `last` follows the rest parameter `keys`, which would cause a syntax error.

The second restriction is that rest parameters cannot be used in an object literal setter. That means this code would also cause a syntax error:

```
let object = {

    // Syntax error: Can't use rest param in setter
    set(...value) {
        // do something
    }
};
```

This restriction exists because object literal setters are restricted to a single argument. Rest parameters are, by definition, an infinite number of arguments, so they're not allowed in this context.

How Rest Parameters Affect the arguments Object

Rest parameters were designed to replace `arguments` in ECMAScript. Originally, ECMAScript 4 did away with `arguments` and added rest parameters to allow an unlimited number of arguments to be passed to functions. ECMAScript 4 never came into being, but this idea was kept around and reintroduced in ECMAScript 6, despite `arguments` not being removed from the language.

The `arguments` object works together with rest parameters by reflecting the arguments that were passed to the function when called, as illustrated in this program:

```
function checkArgs(...args) {
    console.log(args.length);
    console.log(arguments.length);
    console.log(args[0], arguments[0]);
    console.log(args[1], arguments[1]);
}

checkArgs("a", "b");
```

The call to checkArgs() outputs:

```
2
2
a a
b b
```

The arguments object always correctly reflects the parameters that were passed into a function regardless of rest parameter usage.

That’s all you really need to know about rest parameters to get started using them. The next section continues the parameter discussion with the spread operator, which is closely related to rest parameters.

Increased Capabilities of the Function Constructor

The Function constructor is an infrequently used part of JavaScript that allows you to dynamically create a new function. The arguments to the constructor are the parameters for the function and the function body, all as strings. Here’s an example:

```
var add = new Function("first", "second", "return first + second");

console.log(add(1, 1));     // 2
```

ECMAScript 6 augments the capabilities of the Function constructor to allow default parameters and rest parameters. You need only add an equals sign and a value to the parameter names, as follows:

```
var add = new Function("first", "second = first",
        "return first + second");

console.log(add(1, 1));     // 2
console.log(add(1));        // 2
```

In this example, the parameter second is assigned the value of first when only one parameter is passed. The syntax is the same as for function declarations that don't use Function.

For rest parameters, just add the ... before the last parameter, like this:

```
var pickFirst = new Function("...args", "return args[0]");

console.log(pickFirst(1, 2));   // 1
```

This code creates a function that uses only a single rest parameter and returns the first argument that was passed in.

The addition of default and rest parameters ensures that Function has all of the same capabilities as the declarative form of creating functions.

The Spread Operator

Closely related to rest parameters is the spread operator. While rest parameters allow you to specify that multiple independent arguments should be combined into an array, the spread operator allows you to specify an array that should be split and have its items passed in as separate arguments to a function. Consider the Math.max() method, which accepts any number of arguments and returns the one with the highest value. Here's a simple use case for this method:

```
let value1 = 25,
    value2 = 50;

console.log(Math.max(value1, value2));      // 50
```

When you're dealing with just two values, as in this example, Math.max() is very easy to use. The two values are passed in, and the higher value is returned. But what if you've been tracking values in an array, and now you want to find the highest value? The Math.max() method doesn't allow you to pass in an array, so in ECMAScript 5 and earlier, you'd be stuck either searching the array yourself or using apply() as follows:

```
let values = [25, 50, 75, 100]

console.log(Math.max.apply(Math, values));  // 100
```

This solution works, but using apply() in this manner is a bit confusing. It actually seems to obfuscate the true meaning of the code with additional syntax.

The ECMAScript 6 spread operator makes this case very simple. Instead of calling apply(), you can pass the array to Math.max() directly and prefix it with the same ... pattern used with rest parameters. The JavaScript engine then splits the array into individual arguments and passes them in, like this:

```
let values = [25, 50, 75, 100]

// equivalent to
// console.log(Math.max(25, 50, 75, 100));
console.log(Math.max(...values));           // 100
```

Now the call to Math.max() looks a bit more conventional and avoids the complexity of specifying a this-binding (the first argument to Math.max.apply() in the previous example) for a simple mathematical operation.

You can mix and match the spread operator with other arguments as well. Suppose you want the smallest number returned from Math.max() to be 0 (just in case negative numbers sneak into the array). You can pass that argument separately and still use the spread operator for the other arguments, as follows:

```
let values = [-25, -50, -75, -100]

console.log(Math.max(...values, 0));        // 0
```

In this example, the last argument passed to Math.max() is 0, which comes after the other arguments are passed in using the spread operator.

The spread operator for argument passing makes using arrays for function arguments much easier. You'll likely find it to be a suitable replacement for the apply() method in most circumstances.

In addition to the uses you've seen for default and rest parameters so far, in ECMAScript 6, you can also apply both parameter types to JavaScript's Function constructor.

ECMAScript 6’s name Property

Identifying functions can be challenging in JavaScript given the various ways a function can be defined. Additionally, the prevalence of anonymous function expressions makes debugging a bit more difficult, often resulting in stack traces that are hard to read and decipher. For these reasons, ECMAScript 6 adds the `name` property to all functions.

Choosing Appropriate Names

All functions in an ECMAScript 6 program will have an appropriate value for their `name` property. To see this in action, look at the following example, which shows a function and function expression, and prints the `name` properties for both:

```
function doSomething() {
    // ...
}

var doAnotherThing = function() {
    // ...
};

console.log(doSomething.name);          // "doSomething"
console.log(doAnotherThing.name);       // "doAnotherThing"
```

In this code, `doSomething()` has a `name` property equal to `"doSomething"` because it’s a function declaration. The anonymous function expression `doAnotherThing()` has a `name` of `"doAnotherThing"` because that’s the name of the variable to which it is assigned.

Special Cases of the name Property

While appropriate names for function declarations and function expressions are easy to find, ECMAScript 6 goes further to ensure that all functions have appropriate names. To illustrate this, consider the following program:

```
var doSomething = function doSomethingElse() {
    // ...
};

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

console.log(doSomething.name);       // "doSomethingElse"
console.log(person.sayName.name);    // "sayName"
console.log(person.firstName.name); // "get firstName"
```

In this example, doSomething.name is "doSomethingElse" because the function expression itself has a name, and that name takes priority over the variable to which the function was assigned. The name property of person.sayName() is "sayName", as the value was interpreted from the object literal. Similarly, person.firstName is actually a getter function, so its name is "get firstName" to indicate this difference. Setter functions are prefixed with "set" as well.

There are a couple of other special cases for function names, too. Functions created using bind() will have their names prefixed with "bound" and functions created using the Function constructor have a name of "anonymous", as in this example:

```
var doSomething = function() {
    // ...
};

console.log(doSomething.bind().name);   // "bound doSomething"

console.log((new Function()).name);     // "anonymous"
```

The name of a bound function will always be the name of the function being bound prefixed with the string "bound ", so the bound version of doSomething() is "bound doSomething".

Keep in mind that the value of name for any function does not necessarily refer to a variable of the same name. The name property is meant to be informative, to help with debugging, so there's no way to use the value of name to get a reference to the function.

Clarifying the Dual Purpose of Functions

In ECMAScript 5 and earlier, functions serve the dual purpose of being callable with or without new. When used with new, the this value inside a function is a new object and that new object is returned, as illustrated in this example:

```
function Person(name) {
    this.name = name;
}

var person = new Person("Nicholas");
var notAPerson = Person("Nicholas");

console.log(person);        // "[Object object]"
console.log(notAPerson);    // "undefined"
```

When creating notAPerson, calling Person() without new results in undefined (and sets a name property on the global object in nonstrict mode). The capitalization of Person is the only real indicator that the function is meant to be called using new, as is common in JavaScript programs. This confusion over the dual roles of functions led to some changes in ECMAScript 6.

JavaScript has two different internal-only methods for functions: [[Call]] and [[Construct]]. When a function is called without new, the [[Call]] method is executed, which executes the body of the function as it appears in the code. When a function is called with new, that's when the [[Construct]] method is called. The [[Construct]] method is responsible for creating a new object, called the new target, and then executing the function body with this set to the new target. Functions that have a [[Construct]] method are called constructors.

Keep in mind that not all functions have [[Construct]], and therefore not all functions can be called with new. Arrow functions, discussed in the "Section Name" section on page xx, do not have a [[Construct]] method.

Determining How a Function was Called in ECMAScript 5

The most popular way to determine if a function was called with new (and hence, with constructor) in ECMAScript 5 is to use instanceof, for example:

```
function Person(name) {
    if (this instanceof Person) {
        this.name = name;   // using new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person("Nicholas");  // throws error
```

Here, the this value is checked to see if it’s an instance of the constructor, and if so, execution continues as normal. If this isn’t an instance of Person, then an error is thrown. This works because the [[Construct]] method creates a new instance of Person and assigns it to this. Unfortunately, this approach is not completely reliable because this can be an instance of Person without using new, as in this example:

```
function Person(name) {
    if (this instanceof Person) {
        this.name = name;   // using new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person.call(person, "Michael");    // works!
```

The call to Person.call() passes the person variable as the first argument, which means this is set to person inside of the Person function. To the function, there’s no way to distinguish this from being called with new.

The new.target MetaProperty

To solve this problem, ECMAScript 6 introduces the new.target metaproperty. A metaproperty is a property of a non-object that provides additional information related to its target (such as new). When a function’s [[Construct]] method is called, new.target is filled with the target of the new operator. That target is typically the constructor of the newly created object instance that will become this inside the function body. If [[Call]] is executed, then new.target is undefined.

This new metaproperty allows you to safely detect if a function is called with new by checking whether new.target is defined as follows:

```
function Person(name) {
    if (typeof new.target !== "undefined") {
        this.name = name;   // using new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person.call(person, "Michael");    // error!
```

By using new.target instead of this instanceof Person, the Person constructor is now correctly throwing an error when used without new.

You can also check that new.target was called with a specific constructor. For instance, look at this example:

```
function Person(name) {
    if (typeof new.target === Person) {
        this.name = name;   // using new
    } else {
        throw new Error("You must use new with Person.")
    }
}

function AnotherPerson(name) {
    Person.call(this, name);
}

var person = new Person("Nicholas");
var anotherPerson = new AnotherPerson("Nicholas");  // error!
```

In this code, new.target must be Person in order to work correctly. When new AnotherPerson("Nicholas") is called, new.target is set to AnotherPerson, so the subsequent call to Person.call(this, name) will throw an error even though new.target is defined.

Warning: Using new.target outside of a function is a syntax error.

By adding new.target, ECMAScript 6 helped to clarify some ambiguity around functions calls. Following on this theme, ECMAScript 6 also addresses another previously ambiguous part of the language: declaring functions inside of blocks.

Block-Level Functions

In ECMAScript 3 and earlier, a function declaration occurring inside of a block (a block-level function) was technically a syntax error, but all browsers still supported it. Unfortunately, each browser that allowed the syntax behaved in a slightly different way, so it is considered a best practice to avoid function declarations inside of blocks (the best alternative is to use a function expression).

In an attempt to reign in this incompatible behavior, ECMAScript 5 strict mode introduced an error whenever a function declaration was used inside of a block in this way:

```
"use strict";

if (true) {

    // Throws a syntax error in ES5, not so in ES6
    function doSomething() {
        // ...
    }
}
```

In ECMAScript 5, this code throws a syntax error. In ECMAScript 6, the doSomething() function is considered a block-level declaration and can be accessed and called within the same block in which it was defined. For example:

```
"use strict";

if (true) {

    console.log(typeof doSomething);        // "function"

    function doSomething() {
        // ...
    }

    doSomething();
}

console.log(typeof doSomething);            // "undefined"
```

Block level functions are hoisted to the top of the block in which they are defined, so `typeof doSomething` returns `"function"` even though it appears before the function declaration in the code. Once the `if` block is finished executing, `doSomething()` no longer exists.

Deciding When to Use Block-Level Functions

Block level functions are similar to `let` function expressions in that the function definition is removed once execution flows out of the block in which it's defined. The key difference is that block level functions are hoisted to the top of the containing block. Function expressions that use `let` are not hoisted, as this example illustrates:

```
"use strict";

if (true) {

    console.log(typeof doSomething);        // throws error

    let doSomething = function () {
        // ...
    }

    doSomething();
}

console.log(typeof doSomething);
```

Here, code execution stops when typeof doSomething is executed, because the let statement hasn't been executed yet, leaving doSomething() in the TDZ. Knowing this difference, you can choose whether to use block level functions or let expressions based on whether or not you want the hoisting behavior.

Block-Level Functions in Nonstrict Mode

ECMAScript 6 also allows block-level functions in nonstrict mode, but the behavior is slightly different. Instead of hoisting these declarations to the top of the block, they are hoisted all the way to the containing function or global environment. For example:

```
// ECMAScript 6 behavior
if (true) {

    console.log(typeof doSomething);        // "function"

    function doSomething() {
        // ...
    }

    doSomething();
}

console.log(typeof doSomething);            // "function"
```

In this example, doSomething() is hoisted into the global scope so that it still exists outside of the if block. ECMAScript 6 standardized this behavior to remove the incompatible browser behaviors that previously existed, so all ECMAScript 6 runtimes should behave in the same way.

Allowing block-level functions improves your ability to declare functions in JavaScript, but ECMAScript 6 also introduced a completely new way to declare functions.

Arrow Functions

One of the most interesting new parts of ECMAScript 6 is the arrow function. Arrow functions are, as the name suggests, functions defined with a new syntax that uses an "arrow" (=>). But arrow functions behave differently than traditional JavaScript functions in a number of important ways:

- **No this, super, arguments, and new.target bindings** - The value of this, super, arguments, and new.target inside of the function is by the closest containing nonarrow function. (super is covered in Chapter 4.)

- **Cannot be called with new** - Arrow functions do not have a [[Construct]] method and therefore cannot be used as constructors. Arrow functions throw an error when used with new.
- **No prototype** - since you can't use new on an arrow function, there's no need for a prototype. The prototype property of an arrow function doesn't exist.
- **Can't change this** - The value of this inside of the function can't be changed. It remains the same throughout the entire lifecycle of the function.
- **No arguments object** - Since arrow functions have no arguments binding, you must rely on named and rest parameters to access function arguments..
- **No duplicate named arguments** - arrow functions cannot have duplicate named arguments in strict or nonstrict mode, as opposed to nonarrow functions that cannot have duplicate named arguments only in strict mode.

There are a few reasons for these differences. First and foremost, this binding is a common source of error in JavaScript. It's very easy to lose track of the this value inside a function, which can result in unintended program behavior, and arrow functions eliminate this confusion. Second, by limiting arrow functions to simply executing code with a single this value, JavaScript engines can more easily optimize these operations, unlike regular functions, which might be used as a constructor or otherwise modified.

The rest of the differences are also focused on reducing errors and ambiguities inside of arrow functions. By doing so, JavaScript engines are better able to optimize arrow function execution.

Note: Arrow functions also have a name property that follows the same rule as other functions.

Arrow Function Syntax

The syntax for arrow functions comes in many flavors depending upon what you're trying to accomplish. All variations begin with function arguments, followed by the arrow, followed by the body of the function. Both the arguments and the body can take different forms depending on usage. For example, the following arrow function takes a single argument and simply returns it:

```
var reflect = value => value;

// effectively equivalent to:

var reflect = function(value) {
    return value;
};
```

When there is only one argument for an arrow function, that one argument can be used directly without any further syntax. The arrow comes next and the expression to the right of the arrow is evaluated and returned. Even though there is no explicit `return` statement, this arrow function will return the first argument that is passed in.

If you are passing in more than one argument, then you must include parentheses around those arguments, like this:

```
var sum = (num1, num2) => num1 + num2;

// effectively equivalent to:

var sum = function(num1, num2) {
    return num1 + num2;
};
```

The `sum()` function simply adds two arguments together and returns the result. The only difference between this arrow function and the `reflect()` function is that the arguments are enclosed in parentheses with a comma separating them (like traditional functions).

If there are no arguments to the function, then you must include an empty set of parentheses in the declaration, as follows:

```
var getName = () => "Nicholas";

// effectively equivalent to:

var getName = function() {
    return "Nicholas";
};
```

When you want to provide a more traditional function body, perhaps consisting of more than one expression, then you need to wrap the function body in braces and explicitly define a return value, as in this version of `sum()`:

```
var sum = (num1, num2) => {
    return num1 + num2;
};

// effectively equivalent to:

var sum = function(num1, num2) {
    return num1 + num2;
};
```

You can more or less treat the inside of the curly braces the same as you would in a traditional function, with the exception that `arguments` is not available.

If you want to create a function that does nothing, then you need to include curly braces, like this:

```
var doNothing = () => {};

// effectively equivalent to:

var doNothing = function() {};
```

Curly braces are used to denote the function’s body, which works just fine in the cases you’ve seen so far. But an arrow function that wants to return an object literal outside of a function body must wrap the literal in parentheses. For example:

```
var getTempItem = id => ({ id: id, name: "Temp" });

// effectively equivalent to:

var getTempItem = function(id) {

    return {
        id: id,
        name: "Temp"
    };
};
```

Wrapping the object literal in parentheses signals that the braces are an object literal instead of the function body.

Creating Immediately-Invoked Function Expressions

One popular use of functions in JavaScript is creating immediately-invoked function expressions (IIFEs). IIFEs allow you to define an anonymous function and call it immediately without saving a reference. This pattern comes in handy when you want to create a scope that is shielded from the rest of a program. For example:

```
let person = function(name) {

    return {
        getName: function() {
            return name;
        }
    };

}("Nicholas");

console.log(person.getName());      // "Nicholas"
```

In this code, the IIFE is used to create an object with a getName() method. The method uses the name argument as the return value, effectively making name a private member of the returned object.

You can accomplish the same thing using arrow functions, so long as you wrap the arrow function in parentheses:

```
let person = ((name) => {

    return {
        getName: function() {
            return name;
        }
    };

})("Nicholas");

console.log(person.getName());      // "Nicholas"
```

Note that the parentheses are only around the arrow function definition, and not around ("Nicholas"). This is different from a formal function, where the parentheses can be placed outside of the passed-in parameters as well as just around the function definition.

No this Binding

One of the most common areas of error in JavaScript is the binding of this inside of functions. Since the value of this can change inside a single function depending on the context in which the function is called, it’s possible to mistakenly affect one object when you meant to affect another. Consider the following example:

```
var PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click", function(event) {
            this.doSomething(event.type);     // error
        }, false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

In this code, the object PageHandler is designed to handle interactions on the page. The init() method is called to set up the interactions, and that method in turn assigns an event handler to call this.doSomething(). However, this code doesn’t work exactly as intended.

The call to this.doSomething() is broken because this is a reference to the object that was the target of the event (in this case document), instead of being bound to PageHandler. If you tried to run this code, you’d get an error when the event handler fires because this.doSomething() doesn’t exist on the target document object.

You could fix this by binding the value of this to PageHandler explicitly using the bind() method on the function instead, like this:

```
var PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click", (function(event) {
            this.doSomething(event.type);     // no error
        }).bind(this), false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

Now the code works as expected, but it may look a little bit strange. By calling bind(this), you're actually creating a new function whose this is bound to the current this, which is PageHandler. To avoid creating an extra function, a better way to fix this code is to use an arrow function.

Arrow functions have no this binding, which means the value of this inside an arrow function can only be determined by looking up the scope chain. If the arrow function is contained within a nonarrow function, this will be the same as the containing function; otherwise, this is undefined. Here's one way you could write this code using an arrow function:

```
var PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click",
                event => this.doSomething(event.type), false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

The event handler in this example is an arrow function that calls this.doSomething(). The value of this is the same as it is within init(), so this version of the code works similarly to the one using bind(this). Even though the doSomething() method doesn't return a value, it's still the only statement executed in the function body, and so there is no need to include braces.

Arrow functions are designed to be "throwaway" functions, and so cannot be used to define new types; this is evident from the missing prototype property, which regular functions have. If you try to use the new operator with an arrow function, you'll get an error, as in this example:

```
var MyType = () => {},
    object = new MyType();  // error - you can't use arrow functions with 'ne\
w'
```

In this code, the call to new MyType() fails because MyType is an arrow function and therefore has no [[Construct]] behavior. Knowing that arrow functions cannot be used with new allows JavaScript engines to further optimize their behavior.

Also, since the this value is determined by the containing function in which the arrow function is defined, you cannot change the value of this using call(), apply(), or bind().

Arrow Functions and Arrays

The concise syntax for arrow functions makes them ideal for use with array processing, too. For example, if you want to sort an array using a custom comparator, you'd typically write something like this:

```
var result = values.sort(function(a, b) {
    return a - b;
});
```

That's a lot of syntax for a very simple procedure. Compare that to the more terse arrow function version:

```
var result = values.sort((a, b) => a - b);
```

The array methods that accept callback functions such as sort(), map(), and reduce() can all benefit from simpler arrow function syntax, which changes seemingly complex processes into simpler code.

No arguments Binding

Even though arrow functions don't have their own arguments object, it's possible for them to access the arguments object from a containing function. That arguments object is then available no matter where the arrow function is executed later on.

For example:

```
function createArrowFunctionReturningFirstArg() {
    return () => arguments[0];
}

var arrowFunction = createArrowFunctionReturningFirstArg(5);

console.log(arrowFunction());       // 5
```

Inside createArrowFunctionReturningFirstArg(), the arguments[0] element is referenced by the created arrow function. That reference contains the first argument passed to the createArrowFunctionReturningFirstArg() function. When the arrow function is later executed, it returns 5, which was the first argument passed to createArrowFunctionReturningFirstArg(). Even though the arrow function is no longer in the scope of the function that created it, arguments remains accessible due to scope chain resolution of the arguments identifier.

Identifying Arrow Functions

Despite the different syntax, arrow functions are still functions, and are identified as such. Consider the following code:

```
var comparator = (a, b) => a - b;

console.log(typeof comparator);                 // "function"
console.log(comparator instanceof Function);    // true
```

The console.log() output reveales that both typeof and instanceof behave the same with arrow functions as they do with other functions.

Also like other functions, you can still use call(), apply(), and bind() on arrow functions, although the this-binding of the function will not be affected. Here are some examples:

```
var sum = (num1, num2) => num1 + num2;

console.log(sum.call(null, 1, 2));      // 3
console.log(sum.apply(null, [1, 2]));   // 3

var boundSum = sum.bind(null, 1, 2);

console.log(boundSum());                // 3
```

The sum() function is called using call() and apply() to pass arguments, as you'd do with any function. The bind() method is used to create boundSum(), which has its two arguments bound to 1 and 2 so that they don't need to be passed directly.

Arrow functions are appropriate to use anywhere you're currently using an anonymous function expression, such as with callbacks. The next section covers another major ECMAScript 6 development, but this one is all internal, and has no new syntax.

Tail Call Optimization

Perhaps the most interesting change to functions in ECMAScript 6 is an engine optimization, which changes the tail call system. A tail call is when a function is called as the last statement in another function, like this:

```
function doSomething() {
    return doSomethingElse();   // tail call
}
```

Tail calls as implemented in ECMAScript 5 engines are handled just like any other function call: a new stack frame is created and pushed onto the call stack to represent the function call. That means every previous stack frame is kept in memory, which is problematic when the call stack gets too large.

What's Different?

ECMAScript 6 seeks to reduce the size of the call stack for certain tail calls in strict mode (nonstrict mode tail calls are left untouched). With this optimization, instead of creating a new stack frame for a tail call, the current stack frame is cleared and reused so long as the following conditions are met:

1. The tail call does not require access to variables in the current stack frame (meaning the function is not a closure)
2. The function making the tail call has no further work to do after the tail call returns
3. The result of the tail call is returned as the function value

As an example, this code can easily be optimized because it fits all three criteria:

```
"use strict";

function doSomething() {
    // optimized
    return doSomethingElse();
}
```

This function makes a tail call to doSomethingElse(), returns the result immediately, and doesn't access any variables in the local scope. One small change, not returning the result, results in an unoptimized function:

```
"use strict";

function doSomething() {
    // not optimized - no return
    doSomethingElse();
}
```

Similarly, if you have a function that performs an operation after returning from the tail call, then the function can't be optimized:

```
"use strict";

function doSomething() {
    // not optimized - must add after returning
    return 1 + doSomethingElse();
}
```

This example adds the result of doSomethingElse() with 1 before returning the value, and that's enough to turn off optimization.

Another common way to inadvertently turn off optimization is to store the result of a function call in a variable and then return the result, such as:

```
"use strict";

function doSomething() {
    // not optimized - call isn't in tail position
    var result = doSomethingElse();
    return result;
}
```

This example cannot be optimized because the value of doSomethingElse() isn't immediately returned.

Perhaps the hardest situation to avoid is in using closures. Because a closure has access to variables in the containing scope, tail call optimization may be turned off. For example:

```
"use strict";

function doSomething() {
    var num = 1,
        func = () => num;

    // not optimized - function is a closure
    return func();
}
```

The closure func() has access to the local variable num in this example. Even though the call to func() immediately returns the result, optimization can't occur due to referencing the variable num.

How to Harness Tail Call Optimization

In practice, tail call optimization happens behind-the-scenes, so you don't need to think about it unless you're trying to optimize a function. The primary use case for tail call optimization is in recursive functions, as that is where the optimization has the greatest effect. Consider this function, which computes factorials:

```
function factorial(n) {

    if (n <= 1) {
        return 1;
    } else {

        // not optimized - must multiply after returning
        return n * factorial(n - 1);
    }
}
```

This version of the function cannot be optimized, because multiplication must happen after the recursive call to factorial(). If n is a very large number, the call stack size will grow and could potentially cause a stack overflow.

In order to optimize the function, you need to ensure that the multiplication doesn't happen after the last function call. To do this, you can use a default parameter to move the multiplication operation outside of the return statement. The resulting function carries along the temporary result into the next iteration, creating a function that behaves the same but can be optimized by an ECMAScript 6 engine. Here's the new code:

```
function factorial(n, p = 1) {

    if (n <= 1) {
        return 1 * p;
    } else {
        let result = n * p;

        // optimized
        return factorial(n - 1, result);
    }
}
```

In this rewritten version of factorial(), a second argument p is added as a parameter with a default value of 1. The p parameter holds the previous multiplication result so that the next result can be computed without another function call. When n is greater than 1, the multiplication is done first and then passed in as the second argument to factorial(). This allows the ECMAScript 6 engine to optimize the recursive call.

Tail call optimization is something you should think about whenever you're writing a recursive function, as it can provide a significant performance improvement, especially when applied in a computationally-expensive function.

Summary

Functions haven't undergone a huge change in ECMAScript 6, but rather, a series of incremental changes that make them easier to work with.

Default function parameters allow you to easily specify what value to use when a particular argument isn't passed. Prior to ECMAScript 6, this would require some extra code inside the function, to both check for the presence of arguments and assign a different value.

Rest parameters allow you to specify an array into which all remaining parameters should be placed. Using a real array and letting you indicate which parameters to include makes rest parameters a much more flexible solution than arguments.

The spread operator is a companion to rest parameters, allowing you to deconstruct an array into separate parameters when calling a function. Prior to ECMAScript 6, there were only two ways to pass individual parameters contained in an array: by manually specifying each parameter or using apply(). With the spread operator, you can easily pass an array to any function without worrying about the `this` binding of the function.

The addition of the `name` property should help you more easily identify functions for debugging and evaluation purposes. Additionally, ECMAScript 6 formally defines the behavior of block-level functions so they are no longer a syntax error in strict mode.

In ECMAScript 6, the behavior of a function is defined by `[[Call]]`, normal function execution, and `[[Construct]]`, when a function is called with `new`. The `new.target` metaproperty also allows you to determine if a function was called using `new` or not.

The biggest change to functions in ECMAScript 6 was the addition of arrow functions. Arrow functions are designed to be used in place of anonymous function expressions. Arrow functions have a more concise syntax, lexical `this` binding, and no `arguments` object. Additionally, arrow functions can't change their `this` binding, and so can't be used as constructors.

Tail call optimization allows some function calls to be optimized in order to keep a smaller call stack, use less memory, and prevent stack overflow errors. This optimization is applied by the engine automatically when it is safe to do so, however, you may decide to rewrite recursive functions in order to take advantage of this optimization.

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 developers are writing 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, 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 expression 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.

> 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 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. The name change from `mixin()` to `assign()` reflects the actual operation that occurs. Since the `mixin()` method 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. Note, however, that `Object.assign()` does not copy properties whose keys are symbols, which I cover in Chapter 6.

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.

Working with Accessor Properties

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:

```
var receiver = {},
    supplier = {
        get name() {
            return "file.js"
        }
    };

Object.assign(receiver, supplier);

var descriptor = Object.getOwnPropertyDescriptor(receiver, "name");

console.log(descriptor.value);      // "file.js"
console.log(descriptor.get);        // undefined
```

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).

> ⚠ 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 though 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, person.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 in ECMAScript 5:

```
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 it’s 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() {
        return super.getGreeting() + ", hi!";
    }
};

friend.getGreeting();       // throws error!
```

This example uses a named property with a function, and the call to friend.getGreeting() throws an 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. If a function has no [[HomeObject]], or has a different [[HomeObject]] than expected, then this process won't work and an error is thrown, as in this code snippet:

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

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

function getGlobalGreeting() {
    return super.getGreeting() + ", yo!";
}

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

getGlobalGreeting();                // throws error
```

Calling `friend.getGreeting()` returns a string, while calling `getGlobalGreeting()` throws an error for improper use of the `super` keyword. Since the `getGlobalGreeting()` function has no `[[HomeObject]]`, it's not possible to perform a lookup.

Interestingly, the situation doesn't change if `getGlobalGreeting()` is later assigned as a method on the `friend` object, like this:

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

function getGlobalGreeting() {
    return super.getGreeting() + ", yo!";
}

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

// assign getGreeting to the global function
friend.getGreeting = getGlobalGreeting;

friend.getGreeting();               // throws error
```

Here the `getGlobalGreeting()` function overwrites the previously-defined `getGreeting()` method on the `friend` object. Calling `friend.getGreeting()` at that point results in an error as well, because it's now calling the `getGlobalGreeting()` method, which does not have a `[[HomeObject]]`. The value of `[[HomeObject]]` is only set when the function is first created, so even assigning the method onto an object doesn't fix the problem.

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. It can be used either standalone as a method, such as `super()`, or as a reference to the prototype itself, such as `super.getGreeting()`. In both cases, the `this`-binding is set up automatically to work with the current value of `this`.

Destructuring for Easier Data Access

Object and array literals are two of the most frequently used notations in JavaScript, and thanks the popular JSON data format, they've become a particularly important part of the language. It's quite common to define objects and arrays, and then systematically pull out relevant pieces of information from those structures. ECMAScript 6 simplifies this task by adding destructuring, which is the process of breaking a data structure down into smaller parts. This chapter shows you how to harness destructuring for both objects and arrays.

Why is Destructuring Useful?

In ECMAScript 5 and earlier, the need to fetch information from objects and arrays could lead to a lot of code that looks the same, just to get certain data into local variables. For example:

```
let options = {
        repeat: true,
        save: false
    };

// extract data from the object
let repeat = options.repeat,
    save = options.save;
```

This code extracts the values of `repeat` and `save` from the `options` object and stores that data in local variables with the same names. While this code looks simple, imagine if you had a large number of variables to assign; you would have to assign them all one by one. And if there was a nested data structure to traverse to find the information instead, you might have to dig through the entire structure just to find one piece of data.

That's why ECMAScript 6 adds destructuring for both objects and arrays. When you break a data structure into smaller parts, getting the information you need out of it becomes much easier. Many languages implement destructuring with a minimal amount of syntax to make the process simpler to use. The ECMAScript 6 implementation actually makes use of syntax you're already familiar with: the syntax for object and array literals.

Object Destructuring

Object destructuring syntax uses an object literal on the left side of an assignment operation. For example:

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type, name } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

In this code, the value of `node.type` is stored in a variable called `type` and the value of `node.name` is stored in a variable called `name`. This syntax is the same as the object literal property initializer shorthand introduced in Chapter 4. The identifiers `type` and `name` are both declarations of local variables and the properties to read the value from on the `options` object.

Don't Forget the Initializer

When using destructuring to declare variables using `var`, `let`, or `const`, you must supply an initializer (the value after the equals sign). The following lines of code will all throw syntax errors due to a missing initializer:

```
// syntax error!
var { type, name };

// syntax error!
let { type, name };

// syntax error!
const { type, name };
```

While const always requires an initializer, even when using nondestructured variables, var and let only require initializers when using destructuring.

Destructuring Assignment

The object destructuring examples so far have used variable declarations. However, it's also possible to use destructuring in assignments. For instance, you may decide to change the values of variables after they are defined, as follows:

```
let node = {
        type: "Identifier",
        name: "foo"
    },
    type = "Literal",
    name = 5;

// assign different values using structuring
({ type, name } = node);

console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

In this example, type and name are initialized with values when declared, and then two variables with the same names are initialized with different values. The next line uses destructuring assignment to change those values by reading from the node object. Note that you must put parentheses around a destructuring assignment statement. That's because an opening curly brace is expected to a be a block statement, and a block statement cannot appear on the left side of an assignment. The parentheses signal that the next curly brace is not a block statement and should be interpreted as an expression, allowing the assignment to complete.

An error is thrown when the right side of the destructured assignment expression (the expression after =) evaluates to null or undefined. This happens because any attempt to read a property of null or undefined results in a runtime error.

Default Values

When you use a destructuring assignment statement, if you specify a local variable with a property name that doesn't exist on the object, then that local variable is assigned a value of undefined. For example:

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type, name, value } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
console.log(value);     // undefined
```

This code defines an additional local variable called value and attempts to assign it a value. However, there is no corresponding value property on the node object, so the variable is assigned the value of undefined as expected.

You can optionally define a default value to use when a specified property doesn't exist. To do so, insert an equals sign (=) after the property name and specify the default value, like this:

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type, name, value = true } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
console.log(value);     // true
```

In this example, the variable value is given true as a default value. The default value is only used if the property is missing on node or has a value of undefined. Since there is no node.value property, the variable value uses the default value. This works similarly to the default parameter values for functions, as discussed in Chapter 3.

Assigning to Different Local Variable Names

Up to this point, each example destructuring assignment has used the object property name as the local variable name; for example, the value of node.type was stored in a type variable. That works well when you want to use the same name, but what if you don't? ECMAScript 6 has an extended syntax that allows you to assign to a local variable with a different name, and that syntax looks like the object literal nonshorthand property initializer syntax. Here's an example:

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type: localType, name: localName } = node;

console.log(localType);     // "Identifier"
console.log(localName);     // "foo"
```

This code uses destructuring assignment to declare the localType and localName variables, which contain the values from the node.type and node.name properties, respectively. The syntax type: localType says to read the property named type and store its value in the localType variable. This syntax is effectively the opposite of traditional object literal syntax, where the name is on the left of the colon and the value is on the right. In this case, the name is on the right of the colon and the location of the value to read is on the left.

You can add default values when using a different variable name, as well. The equals sign and default value are still placed after the local variable name. For example:

```
let node = {
        type: "Identifier"
    };

let { type: localType, name: localName = "bar" } = node;

console.log(localType);     // "Identifier"
console.log(localName);     // "bar"
```

Here, the localName variable has a default value of "bar". The variable is assigned its default value because there's no node.name property.

So far, you've seen how to deal with destructuring of an object whose properties are primitive values. Object destructuring can also be used to retrieve values in nested object structures.

Nested Object Destructuring

By using a syntax similar to object literals, you can navigate into a nested object structure to retrieve just the information you want. Here's an example:

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
                column: 1
            },
            end: {
                line: 1,
                column: 4
            }
        }
    };

let { loc: { start }} = node;

console.log(start.line);        // 1
console.log(start.column);      // 1
```

The destructuring pattern in this example uses curly braces to indicate that the pattern should descend into the property named `loc` on `node` and look for the `start` property. Remember from the last section that whenever there's a colon in a destructuring pattern, it means the identifier before the colon is giving a location to inspect, and the right side assigns a value. When there's a curly brace after the colon, that indicates that the destination is nested another level into the object.

You can go one step further and use a different name for the local variable as well:

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
                column: 1
            },
            end: {
                line: 1,
                column: 4
            }
        }
    };

// extract node.loc.start
let { loc: { start: localStart }} = node;

console.log(localStart.line);   // 1
console.log(localStart.column); // 1
```

In this version of the code, `node.loc.start` is stored in a new local variable called `localStart`. Destructuring patterns can be nested to an arbitrary level of depth, with all capabilities available at each level.

Object destructuring is very powerful and has a lot of options, but array destructuring offers some unique capabilities that allow you to extract information from arrays.

Syntax Gotcha

Be careful when using nested destructuring because you can inadvertently create a statement that has no effect. Empty curly braces are legal in object destructuring, however, they don't do anything. For example:

```
// no variables declared!
let { loc: {} } = node;
```

There are no bindings declared in this statement. Due to the curly braces on the right, `loc` is used as a location to inspect rather than a binding to create. In such a case, it's likely that the intent was to use `=` to define a default value rather than `:` to define a location. It's possible that this syntax will be made illegal in the future, but for now, this is a gotcha to look out for.

Array Destructuring

Array destructuring syntax is very similar to object destructuring; it just uses array literal syntax instead of object literal syntax. The destructuring operates on positions within an array, rather than the named properties that are available in objects. For example:

```
let colors = [ "red", "green", "blue" ];

let [ firstColor, secondColor ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

Here, array destructuring pulls out the values "red" and "green" from the colors array and stores them in the firstColor and secondColor variables. Those values are chosen because of their position in the array; the actual variable names could be anything. Any items not explicitly mentioned in the destructuring pattern are ignored. Keep in mind that the array itself isn’t changed in any way.

You can also omit items in the destructuring pattern and only provide variable names for the items you’re interested in. If, for example, you just want the third value of an array, you don’t need to supply variable names for the first and second items. Here’s how that works:

```
let colors = [ "red", "green", "blue" ];

let [ , , thirdColor ] = colors;

console.log(thirdColor);        // "blue"
```

This code uses a destructuring assignment to retrieve the third item in colors. The commas preceding thirdColor in the pattern are placeholders for the array items that come before it. By using this approach, you can easily pick out values from any number of slots in the middle of an array without needing to provide variable names for them.

⚠ Similar to object destructuring, you must always provide an initializer when using array destructuring with var, let, or const.

Destructuring Assignment

You can use array destructuring in the context of an assignment, but unlike object destructuring, there is no need to wrap the expression in parentheses. For example:

```
let colors = [ "red", "green", "blue" ],
    firstColor = "black",
    secondColor = "purple";

[ firstColor, secondColor ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

The destructured assignment in this code works in a similar manner to the last array destructuring example. The only difference is that firstColor and secondColor have already been defined. Most of the time, that's probably all you'll need to know about array destructuring assignment, but there's a little bit more to it that you will probably find useful.

Array destructuring assignment has a very unique use case that makes it easier to swap the values of two variables. Value swapping is a common operation in sorting algorithms, and the ECMAScript 5 way of swapping variables involves a third, temporary variable, as in this example:

```
// Swapping variables in ECMAScript 5
let a = 1,
    b = 2,
    tmp;

tmp = a;
a = b;
b = tmp;

console.log(a);     // 2
console.log(b);     // 1
```

The intermediate variable tmp is necessary in order to swap the values of a and b. Using array destructuring assignment, however, there's no need for that extra variable. Here's how you can swap variables in ECMAScript 6:

```
// Swapping variables in ECMAScript 6
let a = 1,
    b = 2;

[ a, b ] = [ b, a ];

console.log(a);     // 2
console.log(b);     // 1
```

The array destructuring assignment in this example looks like a mirror image. The left side of the assignment (before the equals sign) is a destructuring pattern just like those in the other array destructuring examples. The right side is an array literal that is temporarily created for the swap. The destructuring happens on the temporary array, which has the values of b and a copied into its first and second positions. The effect is that the variables have swapped values.

Like object destructuring assignment, an error is thrown when the right side of an array destructured assignment expression evaluates to null or undefined.

Default Values

Array destructuring assignment allows you to specify a default value for any position in the array, too. The default value is used when the property at the given position either doesn't exist or has the value undefined. For example:

```
let colors = [ "red" ];

let [ firstColor, secondColor = "green" ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

In this code, the colors array has only one item, so there is nothing for secondColor to match. Since there is a default value, secondColor is set to "green" instead of undefined.

Nested Destructuring

You can destructure nested arrays in a manner similar to destructuring nested objects. By inserting another array pattern into the overall pattern, the destructuring will descend into a nested array, like this:

```
let colors = [ "red", [ "green", "lightgreen" ], "blue" ];

// later

let [ firstColor, [ secondColor ] ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

Here, the secondColor variable refers to the "green" value inside the colors array. That item is contained within a second array, so the extra square brackets around secondColor in the destructuring pattern are necessary. As with objects, you can nest arrays arbitrarily deep.

Rest Items

Chapter 3 introduced rest parameters for functions, and array destructuring has a similar concept called rest items. Rest items use the ... syntax to assign the remaining items in an array to a particular variable. Here's an example:

```
let colors = [ "red", "green", "blue" ];

let [ firstColor, ...restColors ] = colors;

console.log(firstColor);        // "red"
console.log(restColors.length); // 2
console.log(restColors[0]);     // "green"
console.log(restColors[1]);     // "blue"
```

The first item in colors is assigned to firstColor, and the rest are assigned into a new restColors array. The restColors array, therefore, has two items: "green" and "blue". Rest items are useful for extracting certain items from an array and keeping the rest available, but there's another helpful use.

A glaring omission from JavaScript arrays is the ability to easily create a clone. In ECMAScript 5, developers frequently used the concat() method as an easy way to clone an array. For example:

```
// cloning an array in ECMAScript 5
var colors = [ "red", "green", "blue" ];
var clonedColors = colors.concat();

console.log(clonedColors);      //"[red,green,blue]"
```

While the concat() method is intended to concatenate two arrays together, calling it without an argument returns a clone of the array. In ECMAScript 6, you can use rest items to achieve the same thing through syntax intended to function that way. It works like this:

```
// cloning an array in ECMAScript 6
let colors = [ "red", "green", "blue" ];
let [ ...clonedColors ] = colors;

console.log(clonedColors);      //"[red,green,blue]"
```

In this example, rest items are used to copy values from the colors array into the clonedColors array. While it’s a matter of perception as to whether this techinque makes the developer’s intent clearer than using the concat() method, this is a useful ability to be aware of.

⚠ Rest items must be the last entry in the destructured array and cannot be followed by a comma. Including a comma after rest items is a syntax error.

Mixed Destructuring

Object and array destructuring can be used together to create more complex expressions. In doing so, you are able to extract just the pieces of information you want from any mixture of objects and arrays. For example:

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
                column: 1
            },
            end: {
                line: 1,
                column: 4
            }
        },
        range: [0, 3]
    };

let {
    loc: { start },
    range: [ startIndex ]
} = node;

console.log(start.line);        // 1
console.log(start.column);      // 1
console.log(startIndex);        // 0
```

This code extracts node.loc.start and node.range[0] into start and startIndex, respectively. Keep in mind that loc: and range: in the destructured pattern are just locations that correspond to properties in the node object. There is no part of node that cannot be extracted using destructuring when you use a mix of object and array destructuring. This approach is particularly useful for pulling values out of JSON configuration structures without navigating the entire structure.

Destructured Parameters

Destructuring has one more particularly helpful use case, and that is when passing function arguments. When a JavaScript function takes a large number of optional parameters, one common pattern is to create an options object whose properties specify the additional parameters, like this:

```
// properties on options represent additional parameters
function setCookie(name, value, options) {

    options = options || {};

    let secure = options.secure,
        path = options.path,
        domain = options.domain,
        expires = options.expires;

    // code to set the cookie
}

// third argument maps to options
setCookie("type", "js", {
    secure: true,
    expires: 60000
});
```

Many JavaScript libraries contain `setCookie()` functions that look similar to this one. In this function, the `name` and `value` arguments are required, but `secure`, `path`, `domain`, and `expires` are not. And since there is no priority order for the other data, it's efficient to just have an `options` object with named properties, rather than list extra named parameters. This approach works, but now you can't tell what input the function expects just by looking at the function definition; you need to read the function body.

Destructured parameters offer an alternative that makes it clearer what arguments a function expects. A destructured parameter uses an object or array destructuring pattern in place of a named parameter. To see this in action, look at this rewritten version of the `setCookie()` function from the last example:

```
function setCookie(name, value, { secure, path, domain, expires }) {

    // code to set the cookie
}

setCookie("type", "js", {
    secure: true,
    expires: 60000
});
```

This function behaves similarly to the previous example, but now, the third argument uses destructuring to pull out the necessary data. The parameters outside the destructured parameter are clearly expected, and at the same time, it’s clear to someone using `setCookie()` what options are available in terms of extra arguments. And of course, if the third argument is required, the values it should contain are crystal clear. The destructured parameters also act like regular parameters in that they are set to `undefined` if they are not passed.

> Destructured parameters have all of the capabilities of destructuring that you’ve learned so far in this chapter. You can use default values, mix object and array patterns, and use variable names that differ from the properties you’re reading from.

Destructured Parameters are Required

One quirk of using destructured parameters is that, by default, an error is thrown when they are not provided in a function call. For instance, this call to the `setCookie()` function in the last example throws an error:

```
// Error!
setCookie("type", "js");
```

The third argument is missing, and so it evaluates to `undefined` as expected. This causes an error because destructured parameters are really just a shorthand for destructured declaration. When the `setCookie()` function is called, the JavaScript engine actually does this:

```
function setCookie(name, value, options) {

    let { secure, path, domain, expires } = options;

    // code to set the cookie
}
```

Since destructuring throws an error when the right side expression evaluates to `null` or `undefined`, the same is true when the third argument isn’t passed to the `setCookie()` function.

If you want the destructured parameter to be required, then this behavior isn’t all that troubling. But if you want the destructured parameter to be optional, you can work around this behavior by providing a default value for the destructured parameter, like this:

```
function setCookie(name, value, { secure, path, domain, expires } = {}) {

    // ...
}
```

This example provides a new object as the default value for the third parameter. Providing a default value for the destructured parameter means that the secure, path, domain, and expires will all be undefined if the third argument to setCookie() isn't provided, and no error will be thrown.

Default Values for Destructured Parameters

You can specify destructured default values for destructured parameters just as you would in destructured assignment. Just add the equals sign after the parameter and specify the default value. For example:

```
function setCookie(name, value,
    {
        secure = false,
        path = "/",
        domain = "example.com",
        expires = new Date(Date.now() + 360000000)
    }
) {

    // ...
}
```

Each property in the destructured parameter has a default value in this code, so you can avoid checking to see if a given property has been included in order to use the correct value. There are a couple of downsides to this approach, however. First, the function declaration gets quite a bit more complicated than usual. Second, if the destructured parameter is optional, then it still needs to be assigned an object as a default value, otherwise a call like setCookie("type", "js") throws an error. That default object needs to have all the same information as the destructured parameters (with the same defaults, to ensure consistent behavior), like the one in this version of the setCookie() function:

```
function setCookie(name, value,
    {
        secure = false,
        path = "/",
        domain = "example.com",
        expires = new Date(Date.now() + 360000000)
    } = {
        secure: false,
        path: "/",
        domain: "example.com",
        expires: new Date(Date.now() + 360000000)
    }
) {

    // ...
}
```

Now the function declaration is even more complicated. The first object literal is the destructured parameter while the second is the default value. Unfortunately, this leads to a lot of repetition. You can eliminate some of the repetition by extracting the default values into a separate object and using that separate object as part of both the destructuring and the default parameter value:

```
const setCookieDefaults = {
        secure: false,
        path: "/",
        domain: "example.com",
        expires: new Date(Date.now() + 360000000)
    };

function setCookie(name, value,
    {
        secure = setCookieDefault.secure,
        path = setCookieDefault.path,
        domain = setCookieDefault.domain,
        expires = setCookieDefault.expires
    } = setCookieDefaults
) {

    // ...
}
```

In this code, the default values have been placed in a `setCookieDefaults` object. The destructured parameter references that object directly for setting the default value of each binding and also as the overall default parameter value. The fact that handling all defaults can be complicated is an unfortunate side effect of using destructured parameters. On the bright side, however, if a default value needs to change, you can change it once in `setCookieDefaults` and the data will be used in all the correct spots.

Summary

Destructuring makes working with objects and arrays in JavaScript easier. Using the familiar object literal and array literal syntax, you can pick data structures apart to get at just the information you're interested in. Object patterns allow you to extract data from objects while array patterns let you extract data from arrays.

Both object and array destructuring can specify default values for any property or item that is `undefined` and both throw errors when the right side of an assignment evaluates to `null` or `undefined`. You can also navigate deeply nested data structures with object and array destructuring, descending to any arbitrary depth.

Destructuring declarations use `var`, `let`, or `const` to create variables and must always have an initializer. Destructuring assignments are used in place of other assignments and allow you to destructure into object properties and already-existing variables.

Destructured parameters use the destructuring syntax to make "options" objects more transparent when used as function parameters. The actual data you're interested in can be listed out along with other named parameters. Destructured parameters can be array patterns, object patterns, or a mixture, and you can use all of the features of destructuring.

Symbols

ECMAScript 6 symbols began as a way to create private object members, a feature JavaScript developers have long wanted. Any property with a string name was easy to access regardless of the obscurity of the name. The initial "private names" feature aimed to create non-string property names. That way, normal techniques for detecting these private names wouldn't work.

The private names proposal eventually evolved into ECMAScript 6 symbols. While the implementation details remained the same (non-string values for property names), the goal of privacy was dropped. Instead, symbol properties are categorized separately, non-enumerable by default but still discoverable.

Symbols are a new primitive type, joining the existing primitive types: strings, numbers, booleans, null, and undefined. Symbols are unique among JavaScript primitives in that they do not have a literal form (such as true for boolean or 42 for numbers). The ECMAScript 6 standard uses a special notation to indicate symbols, prefixing the identifier with @@, such as @@create. This book uses this same convention for ease of understanding.

⚠ Despite the notation, symbols do not exactly map to strings beginning with “@@”. Don’t try to use string values where symbols are required.

Creating Symbols

You can create a symbol by using the global Symbol function, such as:

```
var firstName = Symbol();
var person = {};

person[firstName] = "Nicholas";
console.log(person[firstName]);     // "Nicholas"
```

In this example, the symbol firstName is created and used to assign a new property on person. That symbol must be used each time you want to access that same property. It’s a good idea to name the symbol variable appropriately so you can easily tell what the symbol represents.

⚠ Because symbols are primitive values, new Symbol() throws an error when called. It is possible to create an instance of Symbol via new Object(yourSymbol), but it’s unclear when this capability would be useful.

The Symbol function accepts an optional argument that is the description of the symbol. The description itself cannot be used to access the property but is used for debugging purposes. For example:

```
var firstName = Symbol("first name");
var person = {};

person[firstName] = "Nicholas";

console.log("first name" in person);        // false
console.log(person[firstName]);             // "Nicholas"
console.log(firstName);                     // "Symbol(first name)"
```

A symbol's description is stored internally in a property called [[Description]]. This property is read whenever the symbol's toString() method is called either explicitly or implicitly (as in this example). It is not otherwise possible to access [[Description]] directly from code. It's recommended to always provide a description to make both reading and debugging symbols easier.

Identifying Symbols

Since symbols are primitive values, you can use the typeof operator to determine if a variable contains a symbol. ECMAScript 6 extends typeof to return "symbol" when used on a symbol. For example:

```
var symbol = Symbol("test symbol");
console.log(typeof symbol);         // "symbol"
```

While there are other indirect ways of determining whether a variable is a symbol, typeof is the most accurate and preferred way of doing so.

Using Symbols

You can use symbols anywhere you would use a computed property name. You've already seen the use of bracket notation in this chapter, but you can use symbols in computed object literal property names as well as with Object.defineProperty(), and Object.defineProperties(), such as:

```
var firstName = Symbol("first name");

// use a computed object literal property
var person = {
    [firstName]: "Nicholas"
};

// make the property read only
Object.defineProperty(person, firstName, { writable: false });

var lastName = Symbol("last name");

Object.defineProperties(person, {
    [lastName]: {
        value: "Zakas",
        writable: false
    }
});

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

This example first uses a computed object literal property to create the `firstName` symbol property. The property is created as nonenumerable, which is different from computed properties created using nonsymbol names. The following line then sets the property to be read only. Later, a readonly `lastName` symbol property is created using `Object.defineProperties()`. A computed object literal property is used once again, but this time, it's part of the second argument to `Object.defineProperties()`.

While symbols can be used in any place that computed property names are allowed, you'll need to have a system for sharing these symbols in order to use them effectively.

Symbol Coercion

Type coercion is a significant part of JavaScript, and there's a lot of flexibility in the language's ability to coerce one data type into another. Symbols, however, are quite inflexible when it comes to coercion because there exists no logical equivalent of a symbol in other types. Specifically, symbols cannot be coerced into strings or numbers so that they cannot accidentally be used as properties that would otherwise be expected to behave as symbols.

The examples in this chapter have used `console.log()` to indicate the output for symbols, and that works because console.log() calls String() on its arguments. You can use String() directly to get the same result. For instance:

```
var uid = Symbol.for("uid"),
    desc = String(uid);

console.log(desc);              // "Symbol(uid)"
```

The String() function calls uid.toString() and the symbol's string description is returned. If you try to concatenate the symbol directly with a string, however, an error is thrown:

```
var uid = Symbol.for("uid"),
    desc = uid + "";            // error!
```

Concatenating uid with an empty string requires that uid first be coerced into a string. An error is thrown when the coercion is detected, preventing its use in this manner.

Similarly, you cannot coerce a symbol to a number. All mathematical operators cause an error when applied to a symbol. For example:

```
var uid = Symbol.for("uid"),
    sum = uid / 1;              // error!
```

This example attempts to divide the symbol by 1, which causes an error. Errors are thrown regardless of the mathematical operator used (logical operators do not throw an error because all symbols are considered equivalent to true, just like any other non-empty value in JavaScript).

Sharing Symbols

You may find that you want different parts of your code to use the same symbols. For example, suppose you have two different object types in your application that should use the same symbol property to represent a unique identifier. Keeping track of symbols across files or large codebases can be difficult and error-prone. That's why ECMAScript 6 provides a global symbol registry that you can access at any point in time.

When you want to create a symbol to be shared, use the Symbol.for() method instead of calling Symbol(). The Symbol.for() method accepts a single parameter, which is a string identifier for the symbol you want to create (this value is also used as the description). For example:

```
var uid = Symbol.for("uid");
var object = {};

object[uid] = "12345";

console.log(object[uid]);       // "12345"
console.log(uid);               // "Symbol(uid)"
```

The Symbol.for() method first searches the global symbol registry to see if a symbol with the key "uid" exists. If so, then it returns the already existing symbol. If no such symbol exists, then a new symbol is created and registered into the global symbol registry using the specified key. The new symbol is then returned. That means subsequent calls to Symbol.for() using the same key will return the same symbol:

```
var uid = Symbol.for("uid");
var object = {
    [uid]: "12345"
};

console.log(object[uid]);       // "12345"
console.log(uid);               // "Symbol(uid)"

var uid2 = Symbol.for("uid");

console.log(uid === uid2);      // true
console.log(object[uid2]);      // "12345"
console.log(uid2);              // "Symbol(uid)"
```

In this example, uid and uid2 contain the same symbol and so they can be used interchangeably. The first call to Symbol.for() creates the symbol and second call retrieves the symbol from the global symbol registry.

Another unique aspect of shared symbols is that you can retrieve the key associated with a symbol in the global symbol registry by using Symbol.keyFor(), for example:

```
var uid = Symbol.for("uid");
console.log(Symbol.keyFor(uid));    // "uid"

var uid2 = Symbol.for("uid");
console.log(Symbol.keyFor(uid2));   // "uid"

var uid3 = Symbol("uid");
console.log(Symbol.keyFor(uid3));   // undefined
```

Notice that both `uid` and `uid2` return the key `"uid"`. The symbol `uid3` doesn't exist in the global symbol registry, so it has no key associated with it and `Symbol.keyFor()` returns `undefined`.

⚠ The global symbol registry is a shared environment, just like the global scope. That means you can't make assumptions about what is or is not already present in that environment. You should use namespacing of symbol keys to reduce the likelihood of naming collisions when using third-party components. For example, jQuery might prefix all keys with `"jquery."`, such as `"jquery.element"`.

Retrieving Object Symbols

You may be familiar with the `Object.keys()` and `Object.getOwnPropertyNames()` methods for retrieving all property names in an object, with the former returning all enumerable property names and the latter returning all properties regardless of enumerability. Neither of these methods return symbol properties to preserve their ECMAScript 5 functionality. Instead, the `Object.getOwnPropertySymbols()` method was added to allow you to retrieve property symbols from an object.

The return value of `Object.getOwnPropertySymbols()` is an array of symbols for own properties. For example:

```
var uid = Symbol.for("uid");
var object = {
    [uid]: "12345"
};

var symbols = Object.getOwnPropertySymbols(object);

console.log(symbols.length);         // 1
console.log(symbols[0]);             // "Symbol(uid)"
console.log(object[symbols[0]]);     // "12345"
```

In this code, object has a single symbol property uid. The array returned from Object.getOwnPropertySymbols() is an array containing just that symbol.

All objects start off with zero own symbol properties, however, objects can inherit symbol properties from their prototypes. ECMAScript 6 predefines several such properties.

Well-Known Symbols

A central theme for both ECMAScript 5 and ECMAScript 6 was exposing and defining some of the "magic" parts of JavaScript - the parts that couldn't be emulated by a developer. ECMAScript 6 follows this tradition by exposing even more of the previously internal-only logic of the language. It does so primarily through the use of symbol prototype properties that define the basic behavior of certain objects.

ECMAScript 6 has predefined symbols called well-known symbols that represent common behaviors in JavaScript that were previously considered internal-only operations. Each well-known symbol is represented by a property on Symbol, such as Symbol.create for the @@create symbol.

The well-known symbols are:

- @@hasInstance - a method used by instanceof to determine an object's inheritance.
- @@isConcatSpreadable - a Boolean value indicating if use with Array.prototype.concat() should flatten the collection's elements.
- @@iterator - a method that returns an iterator (covered in Chapter 7 - Iterators and Generators).
- @@match - a method used by String.prototype.match() to compare strings.
- @@replace - a method used by String.prototype.replace() to replace substrings.

- @@search - a method used by String.prototype.search() to locate substrings.
- @@species - the constructor from which derived objects are made (covered in Chapter 8 - Classes).
- @@split - a method used by String.prototype.split() to split up strings.
- @@toPrimitive - a method that returns a primitive value representation of the object.
- @@toStringTag - a string used by Object.prototype.toString() to create an object description.
- @@unscopables - an object whose properties are the names of object properties that should not be included in a with statement.

Some of the well-known symbols are discussed below while others are discussed throughout the book to keep them in the correct context.

> Overwriting a method defined with a well-known symbol changes an ordinary object to an exotic object because this changes some internal default behavior.

@@hasInstance

The @@hasInstance symbol is the property on functions that determines whether or not a given object is an instance of that function. The symbol is represented in code by Symbol.hasInstance and the symbol property is defined on Function.prototype so that all functions inherit the default behavior for instanceof. The property itself is defined as nonwritable and nonconfigurable as well as nonenumerable to ensure it doesn't get overwritten by mistake. The @@hasInstance method accepts a single argument, the value to check, and returns true if the value is an instance of the function.

To understand how @@hasInstance works, consider the following code:

```
obj instanceof Array;
```

This code is equivalent to:

```
Array[Symbol.hasInstance](obj);
```

Essentially, ECMAScript 6 redefined the instanceof operator as shorthand syntax for this method call. And now that there’s a method call involved, you can actually change how instanceof works.

For instance, suppose you want to define a function that claims no object as an instance. You can do so by hardcoding the return value of @@hasInstance to false, such as:

```
function MyObject() {
    // ...
}

Object.defineProperty(MyObject, Symbol.hasInstance, {
    value: function(v) {
        return false;
    }
});

var obj = new MyObject();

console.log(obj instanceof MyObject);       // false
```

This example uses Object.defineProperty() to overwrite the @@hasInstance method with a new function. (You must use Object.defineProperty() to overwrite a nonwritable property.) The function always returns false, so even though obj is actually an instance of MyObject, the instanceof operator returns false.

Of course, you can also inspect the value and decide whether or not a value should be considered an instance based on any arbitrary condition. For instance, maybe numbers with values between 1 and 100 are to be considered an instance of a special number type:

```
function SpecialNumber() {
    // ...
}

Object.defineProperty(SpecialNumber, Symbol.hasInstance, {
    value: function(v) {
        return (v instanceof Number) && (v >=1 && v <= 100);
    }
});

var two = new Number(2),
    zero = new Number(0);

console.log(two instanceof SpecialNumber);    // true
console.log(zero instanceof SpecialNumber);   // false
```

This code defines a `@@hasInstance` method that returns true if the value is an instance of `Number` and also has a value between 1 and 100. Note that the left operand to `instanceof` must be an object to trigger the call to `@@hasInstance`, as nonobjects cause `instanceof` to simply return `false` all the time. This code allows `SpecialNumber` to claim `two` as an instance even though there is no directly relationship between them.

> You can also overwrite the default `@@hasInstance` for all builtin functions such as `Date` and `Error`. This isn't recommended as the effects on your code can be unexpected and confusing. It's a good idea to only overwrite `@@hasInstance` on your own functions and only when necessary.

@@isConcatSpreadable

JavaScript arrays have a `concat()` method that is designed to concatenate two arrays together, for example:

```
let colors1 = [ "red", "green" ],
    colors2 = colors1.concat([ "blue", "black" ]);

console.log(colors2.length);    // 4
console.log(colors2);           // ["red","green","blue","black"]
```

This code concatenates a new array to the end of colors1 and creates colors2, a single array with all items from both arrays. However, concat() can also accept nonarray arguments and, in that case, those arguments are simply added to the end of the array. For example:

```
let colors1 = [ "red", "green" ],
    colors2 = colors1.concat([ "blue", "black" ], "brown");

console.log(colors2.length);     // 5
console.log(colors2);            // ["red","green","blue","black","brown"]
```

Here, the extra argument "brown" is passed to concat() and it becomes the fifth item in colors2. Why is an array argument treated differently than a string argument? The specification says that arrays are automatically split into their individual items and all other types are not. Prior to ECMAScript 6, there was no way to adjust this behavior.

The @@isConcatSpreadable property is a boolean value indicating that an object has a length property and numeric keys, and that its numeric property values should be added individually to the result of concat(). The symbol is represented in code by Symbol.isConcatSpreadable but unlike other well-known symbols, this symbol property doesn't appear on any standard objects by default. Instead, it's available as a way to augment how concat() works on certain types of objects, effectively short-circuiting the default behavior. That means you can define any type to behave like arrays in concat(), such as:

```
let collection = {
    0: "Hello",
    1: "world",
    length: 2,
    [Symbol.isConcatSpreadable]: true
};

let messages = [ "Hi" ].concat(collection);

console.log(messages.length);     // 3
console.log(messages);            // ["hi","Hello","world"]
```

The collection object in this example is setup to look like an array: it has a length property and two numeric keys. The @@isConcatSpreadable property is set to true to indicate that the property values should be added as individual items to an array. When collection is passed to concat(), the resulting array has "Hello" and "world" as separate items after "hi".

You can also set @@isConcatSpreadable to false on array subclasses to prevent items from being separated using concat(). Subclassing is discussed in Chapter 8.

@@match, @@replace, @@search, and @@split

There has always been a close relationship between strings and regular expressions in JavaScript. The string type, in particular, has several methods accept regular expressions as arguments:

- match(regex) - determines if the given string matches a regular expression
- replace(regex, replacement) - replaces regular expression matches with a replacement
- search(regex) - locates a regular expression match inside the string
- split(regex) - splits a string into an array on a regular expression match

Prior to ECMAScript 6, the way these methods interacted with regular expressions was hidden from developers. That meant there was no way to mimic what regular expressions did using developer-defined objects. ECMAScript 6 defined four symbols that correspond to these four methods, effectively outsourcing the native behavior to the RegExp builtin object.

The @@match, @@replace, @@search, and @@split symbols represent methods on the regular expression argument that should be called on the first argument to the string methods match(), replace(), search(), and split(), respectively. The four symbol properties are defined on RegExp.prototype as the default implementation that the string methods should use. Knowing this, you can create an object to use with the string methods in a way that is similar to regular expressions. To do, you can use the following symbol in code:

- Symbol.match - a function that accepts a string argument and returns an array of matches or null if no match is found.
- Symbol.replace - a function that accepts a string argument and a replacement string, and returns a string.
- Symbol.search - a function that accepts a string argument and returns the

numeric index of the match or -1 if no match is found.

- Symbol.split - a function that accepts a string argument and returns an array containing pieces of the string split on the match.

Here’s an example:

```
// effectively equivalent to /^.{10}$/
let hasLengthOf10 = {
    [Symbol.match] = function(value) {
        return value.length === 10 ? [value.substring(0, 10)] : null;
    },
    [Symbol.replace] = function(value, replacement) {
        return value.length === 10 ? replacement + value.substring(10) : valu\
e;
    },
    [Symbol.search] = function(value) {
        return value.length === 10 ? 0 : -1;
    },
    [Symbol.split] = function(value) {
        return value.length === 10 ? ["", ""] : [value];
    }
};

let message1 = "Hello world",   // 11 characters
    message2 = "Hello John";    // 10 characters


let match1 = message1.match(hasLengthOf10),
    match2 = message2.match(hasLengthOf10);

console.log(match1);            // null
console.log(match2);            // ["Hello John"]

let replace1 = message1.replace(hasLengthOf10),
    replace2 = message2.replace(hasLengthOf10);

console.log(replace1);          // "Hello world"
console.log(replace2);          // "Hello John"

let search1 = message1.search(hasLengthOf10),
    search2 = message2.search(hasLengthOf10);

console.log(search1);           // -1
console.log(search2);          // 0

let split1 = message1.split(hasLengthOf10),
    split2 = message2.split(hasLengthOf10);

console.log(split1);          // ["Hello world"]
console.log(split2);          // ["", ""]
```

Here, hasLengthOf10 is an object intended to work like a regular expression that matches whenever the string length is exactly 10. Each of the four methods is implemented using the appropriate symbols and then the corresponding methods on two strings are called. The first string, message1, has 11 characters and so will not match; the second string, message2, has 10 characters and so will match. Despite not being a regular expression, hasLengthOf10 is passed to each string method and used correctly due to the additional methods.

While this is a simple example, the ability to perform more complex matches than are currently possible with regular expressions opens up a lot of possibilities.

@@toPrimitive

JavaScript frequently attempts to convert objects into primitive values implicitly when certain operations are applied. For instance, when you compare a string to an object using double equals (==), the object is converted into a primitive value before comparing. Exactly what value should be used was previously an internal operation that is exposed in ECMAScript 6 through the @@toPrimitive method.

The @@toPrimitive method is defined on the prototype of each standard type and prescribes the exact behavior. When a primitive conversion is needed, @@toPrimitive is called with a single argument, referred to as hint in the specification. The hint argument is one of the following string values:

- "number" - a number should be returned
- "string" - a string should be returned
- "default" - the operation has no preference as to the type

For most standard objects, the behavior of number mode is:

1. Call valueOf(), and if the result is a primitive value, return it.
2. Otherwise, call toString(), and if the result is a primitive value, return it.
3. Otherwise, throw an error.

Similarly, for most standard objects, the behavior of string mode is:

1. Call `toString()`, and if the result is a primitive value, return it.
2. Otherwise, call `valueOf()`, and if the result is a primitive value, return it.
3. Otherwise, throw an error.

In many cases, standard objects treat default mode as equivalent to number mode (except for `Date`, which treats default mode as equivalent to string mode). By defining `@@toPrimitive`, you can override these default coercion behaviors.

Default mode is only used for `==`, `+`, and when passing a single argument to the `Date` constructor. Most operations use string or number mode.

To override the default behaviors, use `Symbol.toPrimitive` and assign a function as its value. For example:

```
function Temperature(degrees) {
    this.degrees = degrees;
}

Temperature.prototype[Symbol.toPrimitive] = function(hint) {

    switch (hint) {
        case "string":
            return this.degrees + "\u00b0"; // degrees symbol

        case "number":
            return this.degrees;

        case "default":
            return this.degrees + " degrees";
    }
};

var freezing = new Temperature(32);

console.log(freezing + "!");            // "32 degrees!"
console.log(freezing / 2);              // 16
console.log(String(freezing));          // "32  "
```

This example defines a `Temperature` constructor and overrides the default `@@toPrimitive` method on the prototype. A different value is returned depending on `hint`, where string mode returns the temperature with the Unicode degrees symbol, number mode returns just the numeric value, and default mode appends the word "degrees" after the number. Each of the log statements triggers a different mode: the `+` operator triggers default mode, the `/` operator triggers number mode, and the `String()` function triggers string mode. While it's possible to return different values for all three modes, it's much more common to set to the default mode to be the same as string or number mode.

@@toStringTag

One of the most interesting problems in JavaScript has been the availability of multiple global execution environments. This occurs in web browsers when a page includes an iframe, as the page and the iframe each have their own execution environments. In most cases, this isn't a problem, as data can be passed back and forth between the environments with little cause for concern. The problem arises when trying to identify what type of an object you're dealing with.

The canonical example of this is passing an array from the iframe into the containing page or vice-versa. Now in a different execution environment, instanceof Array returns false because the array was created with a constructor from a different environment.

Developers soon found a good way to identify arrays. It was discovered that by calling the standard toString() method on the object, a predictable string was always returned. Thus, many JavaScript libraries began including a function that works similar to this:

```
function isArray(value) {
    return Object.prototype.toString.call(value) === "[object Array]";
}

console.log(isArray([]));    // true
```

This may look a bit roundabout, but in reality it was found to work quite well in all browsers. The toString() method on arrays isn’t very useful for this purpose because it returns a string representation of the items it contains. The toString() method on Object.prototype, however, had this quirk where it included some internally-defined name in the result. By using this method on an object, you could retrieve what the JavaScript environment thought the data type was.

Developers quickly realized that since there was no way to change this behavior, it was possible to use the same approach to distinguish between native objects and those created by developers. The most important case of this was the ECMAScript 5 JSON object.

Prior to ECMAScript 5, many used Douglas Crockford’s json2.js, which created a global JSON object. As browsers started to implement the JSON global object, it became necessary to tell whether the global JSON was provided by the JavaScript environment itself or through some other library. Using the same technique, many created functions like this:

```
function supportsNativeJSON() {
    return typeof JSON !== "undefined" &&
        Object.prototype.toString.call(JSON) === "[object JSON]";
}
```

Here, the same characteristic that allowed developers to identify arrays across iframe boundaries also provided a way to tell if JSON was the native one or not. A non-native JSON object would return [object Object] while the native version returned [object JSON]. From that point on, this approach became the de facto standard for identifying native objects.

ECMAScript 6 explains this behavior through the @@toStringTag symbol. This symbol represents a property on each object that defines what value should be produced when Object.prototype.toString.call() is called on it. So the value returned for arrays is explained by having the @@toStringTag property equal "Array". Likewise, you can define that value for your own objects:

```
function Person(name) {
    this.name = name;
}

Person.prototype[Symbol.toStringTag] = "Person";

var me = new Person("Nicholas");

console.log(me.toString());                         // "[object Person]"
console.log(Object.prototype.toString.call(me));    // "[object Person]"
```

In this example, a @@toStringTag property is defined on Person.prototype to provide the default behavior for creating a string representation. Since Person.prototype inherits Object.prototype.toString(), the value returned from @@toStringTag is also used when calling me.toString(). However, you can still define your own toString() that provides a different behavior without affecting the use of Object.prototype.toString.call():

```
function Person(name) {
    this.name = name;
}

Person.prototype[Symbol.toStringTag] = "Person";

Person.prototype.toString = function() {
    return this.name;
};

var me = new Person("Nicholas");

console.log(me.toString());                         // "Nicholas"
console.log(Object.prototype.toString.call(me));    // "[object Person]"
```

This code defines Person.prototype.toString() to return the value of the name property.

Since Person instances no longer inherit Object.prototype.toString(), calling me.toString() exhibits a different behavior.

All objects inherit @@toStringTag from Object.prototype unless otherwise specified. The default property value is "Object".

There is no restriction on which values can be used for @@toStringTag on developer-defined objects. For example, there's nothing preventing you from using "Array" as the value of @@toStringTag, such as:

```
function Person(name) {
    this.name = name;
}

Person.prototype[Symbol.toStringTag] = "Array";

Person.prototype.toString = function() {
    return this.name;
};

var me = new Person("Nicholas");

console.log(me.toString());                         // "Nicholas"
console.log(Object.prototype.toString.call(me));    // "[object Array]"
```

Here, the result of calling Object.prototype.toString() is "[object Array]", which is the same as you would get from an actual array. This highlights the fact that Object.prototype.toString() is no longer a completely reliable way of identifying an object's type.

It's possible to change the string tag for native objects by assigning to @@toStringTag on their prototype. For example:

```
Array.prototype[Symbol.toStringTag] = "Magic";

var values = [];

console.log(Object.prototype.toString.call(values));    // "[object Magic]"
```

Even though @@toStringTag is overwritten for arrays in this example, the call to Object.prototype.toString() results in "[object Magic]". While it's recommended not to change built-in objects in this way, there's nothing in the language that forbids it.

@@unscopables

The with statement has long been one of the most controversial parts of JavaScript. Originally designed to avoid repetitive typing, the with statement later became roundly criticized for making code harder to understand and for negative performance implications. As a result, the with statement is not allowed in strict mode (which also affects classes and modules, which are strict mode by default without an opt-out). While there is no future for the with statement, ECMAScript 6 still supports with in nonstrict mode and, as such, had to find ways to allow code to continue to work properly using with.

To understand the complexity of this task, consider the following code:

```
var values = [1, 2, 3],
    colors = ["red", "green", "blue"],
    color = "black";

with(colors) {
    push(color);
    push(...values);
}

console.log(colors);    // ["red", "green", "blue", "black", 1, 2, 3]
```

In this example, the two calls to push() inside the with statement are equivalent to colors.push() because the with statement added push as a local binding. The color reference refers to the variable created outside of with, as does the values reference.

For ECMAScript 6, a values method was added to arrays (discussed in Chapter 7 - Iterators and Generators). That would mean the values reference in the last example should now refer not the local variables values, but to the array's values method, and that would break the code. This is why the @@unscopables symbol exists.

The @@unscopables symbol is used on Array.prototype to indicate which properties should not create bindings inside of a with statement. When present, @@unscopables is an object whose keys are the identifiers to omit from with statement bindings and whose values are true to enforce the block. Here's the default for arrays:

```
// built into ECMAScript 6 by default
Array.prototype[Symbol.unscopables] = Object.assign(Object.create(null), {
    copyWithin: true,
    entries: true,
    fill: true,
    find, true,
    findIndex: true,
    keys: true,
    values: true
});
```

The @@unscopables object has a null prototype (created by Object.create(null)) and contains all of the new array methods in ECMAScript 6 (these methods are covered in detail in Chapter 7 - Iterators and Generators and Chapter 9 - Arrays). Bindings for these methods are not created inside of a with statement, allowing old code to continue working without any problem.

In general, you shouldn't need to define @@unscopables for your objects unless you use the with statement and are making changes to an existing object in your code base.

Summary

Symbols are a new type of primitive value in JavaScript and are used to create nonenumerable properties that require the symbol to access. While not truly private, these properties are harder to accidentally change or overwrite and are therefore suitable for functionality that needs a level of protection from developers.

You can provide descriptions for symbols that allow for easier identifier of symbol values. There is a global symbol registry that allows for the use of shared symbols in different parts of code by using the same description. In this way, the same symbol can be used for the same reason in multiple places.

Symbols are not returned in methods like Object.keys() or Object.getOwnPropertyNames(), so a new method, Object.getOwnPropertySymbols() was added to allow for retrieval of symbol properties. You can still make changes to symbol properties by using Object.defineProperty() and Object.defineProperties().

Well-known symbols define previously internal-only functionality for standard objects and use globally-available symbol constants, such as Symbol.hasInstance. These symbols use the prefix @@ in the specification and allow developers the opportunity to modify standard object behavior in a variety of ways.

Sets and Maps

For most of JavaScript's history, there has been only one type of collection, represented by the Array type. Arrays are used in JavaScript just like arrays in other languages but are often used as queues and stacks as well. Since arrays only use numeric indices, developers used non-array objects whenever a non-numeric index was necessary. That led to custom implementations of sets and maps.

A set is a list of values that cannot contain duplicates. You typically don't access items in the set like you would items in an array; instead, it's much more common to check the set to see if a value is present. A map is a collection of keys that are mapped to specific values. As such, each item in a map stores two pieces of data and values are retrieved by specifying the key to read from. Maps are frequently used as caches, storing data that is to be quickly retrieved later on.

While ECMAScript 5 didn't formally have sets and maps, developers were able to workaround this limitation using non-array objects.

Sets and Maps in ECMAScript 5

In ECMAScript 5 and earlier, developers mimicked sets and maps by using object properties, such as:

```
let set = Object.create(null);

set.foo = true;

// checking for existence
if (set.foo) {

    // do something
}
```

The variable set in this example is an object with a null prototype, ensuring that there are no inherited properties on the object. This is a common ECMAScript 5 approach that uses object properties as unique values to be checked. Properties are added and set to true so they may easily be used in conditional statements (such as the if statement in this example) to see if the value is present. The only real difference between an object used as a set and an object used as a map is the value being stored. For instance:

```
let map = Object.create(null);

map.foo = "bar";

// retrieving a value
let value = map.foo;

console.log(value);         // "bar"
```

This example uses an object as a map, storing a string value "bar" under the key "foo". Unlike sets, maps are mostly used to retrieve information (rather than just checking for the key's existence).

Problems

While the approach of using objects as sets and maps works okay in simple situations, it can get a bit more complicated once you run into the limitations of object properties. Since all object properties must be strings, you must certain that no two keys evaluate to the same string. Consider the following:

```
let map = Object.create(null);

map[5] = "foo";

console.log(map["5"]);      // "foo"
```

This example sets a numeric property of 5 to the value "foo". Internally, that numeric value is converted to a string, so map["5"] and map[5] actually reference the same property. That can cause problems when keys can be either numbers or strings. Another problem is when using objects as keys:

```
let map = Object.create(null),
    key1 = {},
    key2 = {};

map[key1] = "foo";

console.log(map[key2]);      // "foo"
```

Here, map[key2] is referencing the same value as map[key1]. Each object, key1 and key2, are converted to strings because object properties must be strings. The default string representation for objects is "[object Object]", so both key1 and key2 are converted to that string.

Another problem relates specifically to maps with a key whose value is falsy. A falsy value is automatically converted to false when used in situations where a boolean value is required, such as in the condition of an if statement. This alone isn't a problem so long as you're careful as to how you use values. For instance:

```
let map = Object.create(null);

map.count = 1;

// checking for the existence of "count" or a nonzero value?
if (map.count) {
    // ...
}
```

This code has some ambiguity as to the usage of map.count, is the if statement intended to check for the existence of map.count or that the value is nonzero? The code inside the if statement will execute because the value 1 is truthy. However, if map.count is 0, or if map.count doesn't exist, the code inside the if statement would not be executed.

These are difficult problem to identify and debug when it occurs in large applications, and a prime reason that ECMAScript 6 adds both sets and maps to the language.

Sets in ECMAScript 6

ECMAScript 6 adds a Set type that is an ordered list of values without duplicates. Sets allow fast access to the data contained within, adding a more efficient manner of tracking discrete values. Sets are created using new Set() and items are added to the set by using the add() method. You can see how many items are in the set by using the size property.

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.size);    // 2
```

Sets do not coerce values to determine they are the same. So, a set can contain both the number 5 and the string "5" (internally, the comparison is done using

Object.is(), which was discussed in Chapter 4). That means you can also add multiple objects to the set and they remain distinct:

```
let set = new Set(),
    key1 = {},
    key2 = {};

set.add(key1);
set.add(key2);

console.log(set.size);    // 2
```

Because key1 and key2 are not converted to strings, they count as two unique items in the set.

If the add() method is called more than once with the same value, all calls after the first one are effectively ignored:

```
let set = new Set();
set.add(5);
set.add("5");
set.add(5);     // duplicate - this is ignored

console.log(set.size);    // 2
```

You can initialize the set using an array, and the Set constructor will ensure that only unique values are used:

```
let set = new Set([1, 2, 3, 4, 5, 5, 5, 5]);
console.log(set.size);    // 5
```

In this example, an array with feed set is used to initialize the set. The number 5 only appears once in the set even though it appears four times in the array. This functionality makes it easy to convert existing code or JSON structures to use sets.

The Set constructor actually accepts any iterable object as an argument. Arrays work because they are iterable by default, as are sets and maps. The Set constructor uses an iterator to extract values from the argument. Iterables and iterators are discussed in Chapter 8.

You can test to see which values are in the set using the has() method:

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.has(5));    // true
console.log(set.has(6));    // false
```

It's possible to remove a single value from a set by using the delete() method or you can remove all values from the set by using clear():

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.has(5));    // true

set.delete(5);

console.log(set.has(5));    // false
console.log(set.size);      // 1

set.clear();

console.log(set.has("5"));  // false
console.log(set.size);      // 0
```

All of this amounts to a very easy mechanism for tracking unique ordered values. However, what if you want to add items into a set and then perform some operation on each item? That's where the forEach() method comes in.

The forEach() Method

If you’re using to working with arrays, then you may already be familiar with the forEach() method. ECMAScript 5 added forEach() to arrays to create an easier way to work on each item in an array without setting up a for loop. The method proved to be popular amongst developers and so the same method is available on sets and works the same way.

The forEach() method is passed a callback function that accepts three arguments:

1. The value from the next position in the set
2. The same value as the first argument
3. The set from which the value is read

The strange difference between the set version of forEach() and the array version is that the first and second arguments to the callback function are the same. While this might look like a mistake, there’s a good reason for this behavior.

The other objects that have forEach() methods, arrays and maps, pass three arguments to their callback functions. The first two arguments for arrays and maps are the value and the key (the numeric index for arrays). Sets do not have keys, so you could either make the callback function accept two arguments (which would make it different from the others) or find a way to keep the callback function the same and accept three arguments. The decision was made to do the latter, and so sets consider each value to be both the key and the value. As such, the first and second argument are always the same in forEach() on sets in order to keep this functionality consistent with the other forEach() methods on arrays and maps.

Other than the difference in arguments, using forEach() is basically the same for a set as it is for an array:

```
let set = new Set([1, 2]);

set.forEach(function(value, key, ownerSet) {
    console.log(key + " " + value);
    console.log(ownerSet === set);
});
```

This outputs:

```
1 1
true
2 2
true
```

Also the same as arrays, you can pass a `this` value as the second argument to `forEach()` if you need to use `this` in your callback function:

```
let set = new Set([1, 2]),
    processor = {
        output(value) {
            console.log(value);
        },
        process(dataSet) {
            dataSet.forEach(function(value) {
                this.output(value);
            }, this);
        }
    };

processor.process(set);
```

In this example, the `processor.process()` method calls `forEach()` on the set and passes `this` as the `this` value for the callback. That's necessary so `this.output()` will correctly resolve to `processor.output()`. You can also use an arrow function to get the same effect without passing the second argument:

```
let set = new Set([1, 2]),
    processor = {
        output(value) {
            console.log(value);
        },
        process(dataSet) {
            dataSet.forEach((value) => this.output(value));
        }
    };

processor.process(set);
```

The arrow function in this example reads `this` from the containing function and so it correctly resolves `this.output()` to `processor.output()`.

Keep in mind that while sets are great for tracking values and `forEach()` lets you work on each value sequentially, you can't randomly access a value contained in a set like you can with an array. If you need to do so, then the best option is to convert the set into an array.

Converting to an Array

It's easy to convert an array into a set because you can pass the array to the `Set` constructor. It's also easy to convert a set back into an array using the spread operator. The spread operator (...) was introduced in Chapter 3 as a way to split items in an array into separate function parameters. You can also use the spread operator to work on iterable objects such as sets to convert them into an array. For example:

```
let set = new Set([1, 2, 3, 3, 3, 4, 5]),
    array = [...set];

console.log(array);             // [1,2,3,4,5]
```

Here, a set is initially loaded with an array that contains duplicates. The set removes the duplicates and then the items are placed into a new array using the spread operator. The set itself still contains the same items, it's just that they've been copied to a new array. This approach is useful when you already have an array and want to create an array without duplicates. For example:

```
function eliminateDuplicates(items) {
    return [...new Set(items)];
}

let numbers = [1, 2, 3, 3, 3, 4, 5],
    noDuplicates = eliminateDuplicates(numbers);

console.log(noDuplicates);      // [1,2,3,4,5]
```

In the `eliminateDuplicates()` function, the set is just a temporary intermediary used to filter out duplicate values before creating a new array that has no duplicates.

Weak Sets

The `Set` might alternately be called a strong set because of the way it stores object references. An object stored in an instance of `Set` is effectively the same as storing that object inside of variable, meaning that as long as that reference exists, the variable cannot be garbage collected to free memory. For example:

```
let set = new Set(),
    key = {};

set.add(key);
console.log(set.size);      // 1

// eliminate original reference
key = null;

console.log(set.size);      // 1

// get the original reference back
key = [...set][0];
```

In this example, setting `key` to `null` clears one reference of the object but another remains inside the set. So, you can retrieve that value by converting the set to an array using the spread operator and accessing the first item. That works fine for most uses, but sometimes it's better for references in a set to disappear when all other references disappear. For instance, if your JavaScript is running in a web page and wants to keep track of DOM elements (that might be removed by another script), you don't want your code holding onto the last reference to a DOM element - this is called a memory leak.

A weak set is a type of set that only stores weak object references (they cannot store primitive values). A weak reference to an object is one that does not prevent garbage collection if it is the only remaining reference. Weak sets are created using the `WeakSet` constructor and have `add()`, `has()`, and `delete()` methods. Here's an example:

```
let set = new WeakSet(),
    key = {};

// add the object to the set
set.add(key);

console.log(set.has(key));      // true

set.delete(key);

console.log(set.has(key));      // false
```

Using a weak set is a lot like using a regular set. You can add, remove, and check for references in the weak set. You can also seed a weak set with values by passing an iterable to the constructor:

```
let key1 = {},
    key2 = {},
    set = new WeakSet([key1, key2]);

console.log(set.has(key1));     // true
console.log(set.has(key2));     // true
```

In this example, an array is passed to the WeakSet constructor. Since this array contains two objects, those objects are added into the weak set. Keep in mind that an error is thrown if the array contains any non-object values.

Key Differences

The biggest difference between weak sets and regular sets is the weak reference held to the object value. Here's an example of what that means:

```
let set = new WeakSet(),
    key = {};

// add the object to the set
set.add(key);

console.log(set.has(key));      // true

// remove the last strong reference to key, also removes from weak set
key = null;
```

After this code executes, the reference to key in the weak set is no longer accessible. It is not possible to verify its removal because you would need one reference to that object to pass to has(). This can make testing weak sets a little confusing, however, you can trust that the reference has been properly removed by the JavaScript engine.

From these examples, you can see that weak sets share some characteristics with regular sets. The key differences are:

1. The add(), has(), and delete() methods throw an error when passed a non-object.
2. Weak sets are not iterables and therefore cannot be used in a for-of loop.
3. Weak sets do not expose any iterators (such as keys() and values()), so there is no way to programmatically determine the contents of a weak set.
4. Weak sets do not have a forEach() method.

The seemingly limited functionality of weak sets is necessary in order to properly handle memory. In general, if you only need to track object references, then you should use a weak set instead of a regular set.

Sets give you a new way to handle lists of values, but they aren't useful when you need to associate additional information with those values. That's why ECMAScript 6 also adds maps.

Maps in ECMAScript 6

The ECMAScript 6 `Map` type is an ordered list of key-value pairs where both the key and the value can be of any type. Keys are considered to be the same by using `Object.is()`, so you can have both a key of `5` and a key of `"5"` because they are different types. This is quite different than using object properties as keys, which always coerce values into strings.

Items are added to maps by using the `set()` method and passing in the key and the value to associate with the key. You can later retrieve a value by passing the key to `get()`. For example:

```
let map = new Map();
map.set("title", "Understanding ES6");
map.set("year", 2016);

console.log(map.get("title"));      // "Understanding ES6"
console.log(map.get("year"));       // 2016
```

In this example, two key-value pairs are stored. The key `"title"` stores a string while the key `"year"` stores a number. These are later retrieved using `get()`. If the key doesn't exist in the map, then the special value `undefined` is returned when calling `get()`.

You can also use objects as keys, something that isn't possible using object properties. Here's an example:

```
let map = new Map(),
    key1 = {},
    key2 = {};

map.set(key1, 5);
map.set(key2, 42);

console.log(map.get(key1));         // 5
console.log(map.get(key2));         // 42
```

This code uses the objects `key1` and `key2` as keys in the map to store two different values. Because these keys are not coerced into another form, each object is considered unique. This allows you to associate additional data to an object without modifying the object itself.

Map Methods

Maps share several methods with sets, and that is intentional to allow maps and sets to be interacted with in similar ways. These three methods are available on both maps and sets:

- `has(key)` - determines if the given key exists in the map
- `delete(key)` - removes the key and its associated value from the map
- `clear()` - removes all keys and values from the map

Additionally, maps have a `size` property that indicates how many key-value pairs it contains. Here’s an example:

```
let map = new Map();
map.set("name", "Nicholas");
map.set("age", 25);

console.log(map.size);          // 2

console.log(map.has("name"));   // true
console.log(map.get("name"));   // "Nicholas"

console.log(map.has("age"));    // true
console.log(map.get("age"));    // 25

map.delete("name");
console.log(map.has("name"));   // false
console.log(map.get("name"));   // undefined
console.log(map.size);          // 1

map.clear();
console.log(map.has("name"));   // false
console.log(map.get("name"));   // undefined
console.log(map.has("age"));    // false
console.log(map.get("age"));    // undefined
console.log(map.size);          // 0
```

As with sets, the `size` property always contains the number of key-value pairs in the map. This example starts with two keys `"name"` and `"age"`, so `has()` returns `true` when passed either key. The `"name"` is then removed by using `delete()`, so `has()` returns `false` when passed `"name"` and the `size` property indicates one less item. The `clear()` method then removes the remaining key, as indicated by `has()` returning `false` for both keys and `size` being 0.

The `clear()` method is a fast way to remove a lot of data from a map, but there’s also a way to add a lot of data to a map at one time.

Map Initialization

Also similar to sets, you can initialize a map with data by passing an array to the `Map` constructor. Each item in the array must itself be an array where the first item is the key and the second is the value. The entire map, therefore, is an array of these two-item arrays, for example:

```
let map = new Map([ ["name", "Nicholas"], ["age", 25]]);

console.log(map.has("name"));   // true
console.log(map.get("name"));   // "Nicholas"
console.log(map.has("age"));    // true
console.log(map.get("age"));    // 25
console.log(map.size);          // 2
```

The keys "name" and "age" are added into the map through initialization in the constructor. While the array-of-arrays may look a bit strange, it is necessary in order to accurately represent keys as they can be any data type. Storing these keys in an array is the only way to ensure that they are not coerced into another data type before storage.

The forEach Method

The forEach() method for maps is similar to forEach() for sets and arrays in that it accepts a callback function that receives three arguments:

1. The value from the next position in the map
2. The key for that value
3. The map from which the value is read

The callback arguments more closely match the forEach() behavior in arrays where the first argument is the value and the second is the key (numeric index in arrays). Here's an example:

```
let map = new Map([ ["name", "Nicholas"], ["age", 25]]);

map.forEach(function(value, key, ownerMap) {
    console.log(key + " " + value);
    console.log(ownerMap === map);
});
```

This outputs:

```
name Nicholas
true
age 25
true
```

The callback receives each key-value pair in the order in which they were inserted, which is slightly different from arrays, where the callback receives each item in order of numeric index.

You can also provide a second argument to forEach() to specify the this value inside of the callback function. This behaves the same as with the set version of forEach().

Weak Maps

Weak maps are to maps what weak sets are to sets, which is a way to store weak object references. In weak maps, every key must be an object (and error is thrown if you try to use a non-object key), and those object references are held weakly so as not to interfere with garbage collection. When there are no other references to a weak map key, the key-value pair is removed from the weak map.

The best example for using weak maps is to create an object related to a particular DOM element in a web page. For example, some JavaScript libraries for web pages maintain one custom object for every DOM element that is referenced through the library, and that mapping is stored in a cache of objects internally. The difficult part of this approach is determining when a DOM element no longer exists in the web page so that the library can remove its associated object (and most importantly, not hold onto the DOM element reference and cause a memory leak). Using a weak map would allow the library to associate a custom object with every DOM element and then automatically destroy that object when the DOM element no longer exists.

It's important to note that only weak map keys, and not weak map values, are weak references. An object stored as a weak map value will prevent garbage collection if all other references are removed.

Using Weak Maps

The ECMAScript 6 WeakMap type is an unordered list of key-value pairs where the key must be a non-null object and the value can be of any type. The interface for WeakMap is very similar to that of Map in that set() and get() are used to add and retrieve data respectively:

```
let map = new WeakMap(),
    element = document.querySelector(".element");

map.set(element, "Original");

let value = map.get(element);
console.log(value);             // "Original"

// remove the element
element.parentNode.removeChild(element);
element = null;

// the weak map is empty at this point
```

In this example, one key-value pair is stored. The key is a DOM element used to store a corresponding string value. That value is then retrieved by passing in the DOM element to `get()`. If the DOM element is then removed from the document and the variable referencing it is set to `null`, then the data is also removed from the weak map.

Similar to weak sets, there is no way to verify that the weak map is empty because it doesn’t have a `size` property. Because there are no remaining references to the key, you can’t use `get()` to attempt to retrieve the value. The weak map has cut off access to the value for that key and, when the garbage collector runs, that memory will be freed.

Weak Map Initialization

Weak maps can be initialized the same way as regular maps: by passing an array of arrays to the `WeakMap` constructor. Once again, each item in the array should be a two-item array where the first item is the key (an object) and the second item is the value (any data type) . For example:

```
let key1 = {},
    key2 = {},
    map = new WeakMap([ [key1, "Hello"], [key2, 42]]);

console.log(map.has(key1));      // true
console.log(map.get(key1));      // "Hello"
console.log(map.has(key2));      // true
console.log(map.get(key2));      // 42
```

The objects `key1` and `key2` are used as keys in the weak map and they can be accessed using get() and has(). An error is thrown if there is a non-object key in data passed to WeakMap.

Weak Map Methods

Weak maps have only a couple of additional methods available to interact with its items. There is a has() method to determine if the given key exists in the map and a delete() method to remove a specific key-value pair. There is no clear() method because that would require enumerating keys, and like weak sets, that is not possible with weak maps. Here's an example:

```
let map = new WeakMap(),
    element = document.querySelector(".element");

map.set(element, "Original");

console.log(map.has(element));   // true
console.log(map.get(element));   // "Original"

map.delete(element);
console.log(map.has(element));   // false
console.log(map.get(element));   // undefined
```

Here, a DOM element is once again used as the key in a weak map. The has() method is useful for checking to see if a reference is currently being used as a key in the weak map. Keep in mind that this only works when you have a non-null reference to a key. The key is forcibly removed from the weak map by using delete(), at which point has() returns false and get() returns undefined.

Private Object Data

While most developers consider the main use case of weak maps to be associated data with DOM elements, there are many possible uses (and no doubt, some that have yet to be discovered). One practical use of weak maps is to store data that is private to object instances. All object properties are public in ECMAScript 6 and so you need to use some creativity to make data access to objects but not accessible to everything. Consider the following example:

```
function Person(name) {
    this._name = name;
}

Person.prototype.getName = function() {
    return this._name;
};
```

This code uses the common convention of a leading underscore to indicate that a property is considered private and should not be modified outside of the object instance. The intent is to use getName() to read this._name and not allow that value to be changed. However, there is nothing standing in the way of someone writing to the _name property, so it can be overwritten either intentionally or accidentally.

Using ECMAScript 5, it’s possible to get close to having truly private data using a pattern such as:

```
let Person = (function() {

    let privateData = {},
        privateId = 0;

    function Person(name) {
        Object.defineProperty(this, "_id", { value: privateId++ });

        privateData[this._id] = {
            name: name
        };
    }

    Person.prototype.getName = function() {
        return privateData[this._id].name;
    };

    return Person;
}());
```

This example wraps the definition of Person is an IIFE that contains two private variables, privateData and privateId. The privateData object stores private information for each instance while privateId is used to generate a unique ID for each instance. When the Person constructor is called, a _id property is added so that it’s nonenumerable, nonconfigurable, and nonwritable (mean it can’t accidentally be overwritten). Then, a entry is made into the privateData object that corresponds to the ID for the object instance; that's where the name information is stored. Later, in getName(), the name can be restored by using this._id as the key into privateData. Because privateData is not accessible outside of the IIFE, the actual data is safe even though this._id is exposed publicly. The big problem with this approach is that the data in privateData always remains because there is no way to know when an object instance is destroyed, meaning that extra data will always be left in privateData. This problem can be solved by using a weak map instead:

```
let Person = (function() {

    let privateData = new WeakMap();

    function Person(name) {
        privateData.set(this, { name: name });
    }

    Person.prototype.getName = function() {
        return privateData.get(this).name;
    };

    return Person;
}());
```

This version of the code uses a weak map for the private data instead of an object. Because the object instance itself can be used as a key, there's no need to keep track of a separate ID. When the Person constructor is called, a new entry is made into the weak map with a key of this and a value of an object containing private information (in this case, just name). That information is retrieved inside of getName() by passing this to privateData.get() in order to retrieve the data object and access the name property. In this way, the private information is kept private and will be destroyed whenever an object instance is destroyed.

Uses and Limitations

When deciding whether to use a weak map or a regular map, the primary decision is whether you want to use only object keys. Anytime you're going to use only object keys then the best choice is a weak map. That will allow you to optimize memory usage and avoid memory leaks by ensuring that extra data isn't kept around after it's no longer accessible.

Keep in mind that weak maps give you very little visibility into their contents, so you can't use forEach(), size, or clear() to manage the items. If you need some inspection capabilities, then regular maps are a better choice. Just be sure to keep an eye on memory usage.

Of course, if you only want to use non-object keys, then regular maps are your only choice.

Summary

ECMAScript 6 formally introduces sets and maps into the language. Prior to this, developers frequently used objects to mimic both sets and maps, often running into problems due to the limitations associated with object properties.

Sets are unordered lists of unique values. Values are considered unique if they are not equivalent according to `Object.is()`. Sets automatically remove duplicate values, so you can use a set to filter an array for duplicates and return the result. Sets aren't subclasses of arrays, so you cannot randomly access its values. Instead, you can use the `has()` method to determine if a value is contained in the set and the `size` property to inspect the number of values in the set. There's also a `forEach()` method to process each set value.

Weak sets are a special type of set that can contain only objects. The objects are stored with weak references, meaning that they will not block garbage collection if it is the only remaining reference. It's not possible to inspect weak set contents due to the complexities of memory management, so it's best to use them only for tracking objects that need to be grouped together.

Maps are unordered key-value pairs where the key can be any data type. Similar to sets, duplicate keys are determined by `Object.is()`, which means you can have a numeric key `5` and a string `"5"` as two separate keys. Any data type can be used for a value to associate with the key using `set()` and the value can later be retrieved by using `get()`. Maps also have a `size` property and a `forEach()` method to allow for easier item access.

Weak maps are a special type of map that can only have object keys. As with weak sets, these object key references are weak and do no prevent garbage collection when they are the only remaining reference. When a key is garbage collected, the value associated with it is also removed from the weak map. This memory management aspect makes weak maps uniquely suited for correlating additional information with objects whose lifecycles are managed outside of the code accessing them.

Iterators and Generators

Iterators have been used in many programming languages as a way to more easily work with collections of data. Many languages have moved away from needing to use for loops, where initialization of variables that track position in a collection are necessary, to using iterator objects that programmatically return the next item in a collection. In ECMAScript 6, JavaScript adds iterators as an important feature of the language. When coupled with new array methods and new types of collections (such as sets and maps), iterators become even more important for efficient processing of data.

The Loop Problem

If you've ever written JavaScript, chances are you've written some code that looks like this:

```
var colors = ["red", "green", "blue"];

for (var i=0, len < colors.length; i < len; i++) {
    console.log(colors[i]);
}
```

This is a standard for loop that sets up the variable i to track the index into the array. The value of i is incremented each time through the loop if it's not larger than the length of the array (stored in len). While this is a fairly straightforward example, the complexity grows when you nest loops and need to keep track of multiple variables. This additional complexity can lead to errors, and the boilerplate nature of the code lends itself to more errors as similar code is written in multiple places. This is the problem that iterators are meant to solve.

What are Iterators?

Iterators are nothing more than objects with a specific interface. That interface consists of a method called next() that returns a result object. The result object has two properties, value, which is the next value, and done, which is a boolean value that's true when there are no more values to return. The iterator keeps an internal pointer to a location within a collection of values and, with each call to next(), returns the next appropriate value.

If you call next() after the last value has been returned, the method returns done as true and value contains the return value for the iterator. The return value is not considered part of the data set, but rather a final piece of related data or undefined if no such data exists. (This concept will become clearer in the generators section later in this chapter.)

With that understanding, it's fairly easy to create an iterator using ECMAScript 5, for example:

```
function createIterator(items) {

    var i = 0;

    return {
        next: function() {

            var done = (i >= items.length);
            var value = !done ? items[i++] : undefined;

            return {
                done: done,
                value: value
            };

        }
    };
}

var iterator = createIterator([1, 2, 3]);

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"

// for all further calls
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

The `createIterator()` function in this example returns an object with a `next()` method. Each time the method is called, the next value in the `items` array is returned as `value`. When `i` is 3, `items[i++]` returns `undefined` and `done` is `true`, which fulfills the special last case for iterators in ECMAScript 6.

You might be thinking that iterators look interesting but seem complex to write. Indeed, writing iterators so that they adhere to the correct behavior is a bit difficult, which is why ECMAScript 6 provides generators.

Generators

A generator is a special kind of function that returns an iterator. Generator functions are indicated by inserting a star character (*) after the function keyword (it doesn't matter if the star is directly next to function or if there's some whitespace between them). The yield keyword is used inside of generators to specify the values that the iterator should return when next() is called. So if you want to return three different values for each successive call to next(), you can do so as follows:

```
// generator
function *createIterator() {
    yield 1;
    yield 2;
    yield 3;
}

// generators are called like regular functions but return an iterator
let iterator = createIterator();

console.log(iterator.next().value);     // 1
console.log(iterator.next().value);     // 2
console.log(iterator.next().value);     // 3
```

In this example, the createIterator() function is a generator (as indicated by the * before the name) and it's called like any other function. The value returned is an iterator. Multiple yield statements inside the generator indicate the progression of values that should be returned when next() is called on the iterator. First, next() should return 1, then 2, and then 3 before the iterator is finished.

Perhaps the most interesting aspect of generator functions is that they stop execution after each yield statement, so yield 1 executes and then the function doesn't execute anything else until the iterator's next() method is called. At that point, execution resumes with the next statement after yield 1, which in this case is yield 2. This ability to stop execution in the middle of a function is extremely powerful and lends to some interesting uses of generator functions (discussed later in this chapter).

The yield keyword can be used with any value or expression, so you can do interesting things like use yield inside of a for loop:

```
function *createIterator(items) {
    for (let i=0; i < items.length; i++) {
        yield items[i];
    }
}

let iterator = createIterator([1, 2, 3]);

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"

// for all further calls
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

In this example, an array is used in a for loop, yielding each item as the loop progresses. Each time yield is encountered, the loop stops, and each time next() is called on iterator, the loop picks back up where it left off.

Generator functions are an important part of ECMAScript 6, and since they are just functions, they can be used in all the same places.

Generator Function Expressions

Generators can be created using function expressions in the same way as using function declarations by including a star (*) character between the function keyword and the opening paren, for example:

```
let createIterator = function *(items) {
    for (let i=0; i < items.length; i++) {
        yield items[i];
    }
};

let iterator = createIterator([1, 2, 3]);

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"

// for all further calls
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

In this code, `createIterator()` is a generator function expression rather than a function declaration (as in the previous example). Since the function expression is anonymous, the asterisk is between the `function` keyword and the opening parentheses. Otherwise, this example is the same as the previous one using a generator function declaration.

ℹ It is not possible to create an arrow function that is also a generator.

Generator Object Methods

Because generators are just functions, they can be added to objects the same way as any other functions. For example, you can use an ECMAScript 5-style object literal with a function expression:

```
var o = {

    createIterator: function *(items) {
        for (let i=0; i < items.length; i++) {
            yield items[i];
        }
    }
};

let iterator = o.createIterator([1, 2, 3]);
```

You can also use the ECMAScript 6 method shorthand by prepending the method name with a star (*):

```
var o = {

    *createIterator(items) {
        for (let i=0; i < items.length; i++) {
            yield items[i];
        }
    }
};

let iterator = o.createIterator([1, 2, 3]);
```

This example is functionally equivalent to the previous one, the only difference is the syntax used. Because the method is defined using shorthand and there is no `function` keyword, the star is placed immediately before the method name (though there may be whitespace between the star and the method name).

Iterables and for-of

Closely related to the concept of an iterator is an iterable. An iterable is an object with a `@@iterator` symbol property. The well-known `@@iterator` symbol specifies a function that returns an iterator for the given object. All of the collection objects, including arrays, sets, and maps, as well as strings, are iterables in ECMAScript 6 and so have a default iterator specified. Iterables are designed to be used with a new addition to ECMAScript: the `for-of` loop.

All iterators created by generators are also iterables, as generators assign the @@iterator property by default.

The for-of loop is the second part of the solution to the problem introduced at the beginning of this chapter. Instead of requiring you to track an index into a collection, the for-of loop works by calling next() on an iterable each time through the loop and storing the value from the result object in a variable. The loop continues this process until the done property of the returned object is true. For example:

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

for (let num of values) {
    console.log(num);
}
```

This code outputs the following:

```
1
2
3
```

The for-of loop in this example is first calling the @@iterator method of the array to retrieve an iterator (as mentioned earlier, all arrays are iterables). Next, iterator.next() is called and the value property on the result object is read into num. So num is first 1, then 2, and finally 3. When done on the result object is true, the loop exits, so num is never assigned the value of undefined.

The advantage of the for-of loop as opposed to the traditional for loop is that you never have to keep track of an index into a collection. Instead, you can just focus on working with the contents of the collection.

The for-of statement will throw an error when used on, a non-iterable, null, or undefined.

Accessing the Default Iterator

You can access the default iterator for an object using Symbol.iterator, for example:

```
let values = [1, 2, 3];
let iterator = values[Symbol.iterator]();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

This code gets the default iterator for values and uses that to iterate over the values in the array. This is the same process that happens behind-the-scenes when using a for-of loop.

Knowing that Symbol.iterator specifies the default iterator, it's possible to detect if an object is iterable by using the following:

```
function isIterable(object) {
    return typeof object[Symbol.iterator] === "function";
}

console.log(isIterable([1, 2, 3]));     // true
console.log(isIterable("Hello"));       // true
console.log(isIterable(new Map()));     // true
console.log(isIterable(new Set()));     // true
console.log(isIterable(new WeakMap())); // false
console.log(isIterable(new WeakSet())); // false
```

The isIterable() function simply checks to see if a default iterator exists on the object and is a function. This is similar to the check that the for-of loop does before executing.

Now that you know about accessing the default iterator using Symbol.iterator, you can also use that property to create your own iterables.

Creating Iterables

Developer-defined objects are not iterable by default, but you can make them iterable by creating a Symbol.iterator property containing a generator. For example:

```
let collection = {
    items: [],
    *[Symbol.iterator]() {
        for (let item of this.items) {
            yield item;
        }
    }

};

collection.items.push(1);
collection.items.push(2);
collection.items.push(3);

for (let x of collection) {
    console.log(x);
}

// Output:
// 1
// 2
// 3
```

This code defines a default iterator for a variable called collection using object literal method shorthand and a computed property using Symbol.iterator (note the star still comes before the name). The generator then uses a for-of loop to iterate over the values in this.items and uses yield to return each one. So instead of manually iterating defining values to return as part of the default iterator, the collection object relies on the default iterator of this.items to do the work.

> Another approach to using the iterator of another object is to use delegating generators, a topic discussed later in this chapter.

So far you've seen some uses for the default array iterator, but there are many more iterators built in to ECMAScript 6 to make working with collections of data easy.

Built-in Iterators

Iterators are an important part of ECMAScript 6 and, as such, iterators are available on many objects by default. You don’t need to create your own iterators for many of the built-in types because the language has them already. You only need to create iterators when you find that the built-in ones don’t serve your purpose, most frequently when defining your own objects or classes. Otherwise, you can rely on the built-in iterators to do your work. Perhaps the most common iterators to use are those that work on collections.

Collection Iterators

ECMAScript 6 has three types of collection objects: arrays, maps, and sets. All three have the same built-in iterators to help you navigate their content. You can retrieve an iterator for a collection by calling one of these methods:

- `entries()` - returns an iterator whose values are a key-value pair.
- `values()` - returns an iterator whose values are the values of the collection.
- `keys()` - returns an iterator whose values are the keys contained in the collection.

The `entries()` iterator returns a two-item array each time `next()` is called. The two-item array represents the key and value for each item in the collection. For arrays, the first item is the numeric index; for sets, the first item is also the value (since values double as keys in sets); for maps, the first item is the key. Here are some examples:

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ECMAScript 6");
data.set("format", "ebook");

for (let entry of colors.entries()) {
    console.log(entry);
}

for (let entry of tracking.entries()) {
    console.log(entry);
}

for (let entry of data.entries()) {
    console.log(entry);
}
```

This example outputs the following:

```
[0, "red"]
[1, "green"]
[2, "blue"]
[1234, 1234]
[5678, 5678]
[9012, 9012]
["title", "Understanding ECMAScript 6"]
["format", "ebook"]
```

This code uses the `entries()` method on each type of collection to retrieve an iterator and `for-of` loops to iterate the items. From the console output, you can see the keys and values returned in pairs for each object.

The `values()` iterator simply returns the values as they are stored in the collection. For example:

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ECMAScript 6");
data.set("format", "ebook");

for (let value of colors.values()) {
    console.log(value);
}

for (let value of tracking.values()) {
    console.log(value);
}

for (let value of data.values()) {
    console.log(value);
}
```

This example outputs the following:

```
"red"
"green"
"blue"
1234
5678
9012
"Understanding ECMAScript 6"
"ebook"
```

In this case, using `values()` returns the exact data contained in the collection without any information as to its location.

The `keys()` iterator returns each key present in the collection. For arrays, this is the numeric keys only (it never returns other own properties of the array); for sets, the keys are the same as the values and so `keys()` and `values()` return the same iterator; for maps, this is each unique key. Here’s an example:

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ECMAScript 6");
data.set("format", "ebook");

for (let key of colors.keys()) {
    console.log(key);
}

for (let key of tracking.keys()) {
    console.log(key);
}

for (let key of data.keys()) {
    console.log(key);
}
```

This example outputs the following:

```
0
1
2
1234
5678
9012
"title"
"format"
```

When using the `keys()` iterator, you'll receive only the corresponding keys in the collection. For arrays, this means you'll only receive numeric indices even if you've added named properties to array object. This is different from the way the `for-in` loop works with arrays because the `for-in` loop iterates over properties rather than just the numeric indices.

Additionally, each collection type has a default iterator that is used by `for-of` whenever an iterator isn't explicitly specified. The default iterator for arrays and sets is `values()` while the default iterator for maps is `entries()`. This makes it a little bit easier to use collection objects in `for-of`:

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ECMAScript 6");
data.set("format", "ebook");

// same as using colors.values()
for (let value of colors) {
    console.log(value);
}

// same as using tracking.values()
for (let num of tracking) {
    console.log(num);
}

// same as using data.entries()
for (let entry of data) {
    console.log(entry);
}
```

This example outputs the following:

```
"red"
"green"
"blue"
1234
5678
9012
["title", "Understanding ECMAScript 6"]
["format", "ebook"]
```

The default iterators for arrays, sets, and maps are designed to work similarly to how these objects are initialized. So arrays and sets return their values by default while maps return the same array format that can be passed into the `Map` constructor. This is useful when used in `for-of` loops with destructuring, as in this example:

```
let data = new Map();

data.set("title", "Understanding ECMAScript 6");
data.set("format", "ebook");

// same as using data.entries()
for (let [key, value] of data) {
    console.log(key + "=" + value);
}
```

The `for-of` loop in this example uses a destructured array to assign `key` and `value` for each entry in the map. In this way, you can easily work with keys and values at the same time without needing to access a two-item array.

Weak sets and weak maps do not have any built-in iterators. Managing weak references means there’s no way to know exactly how many values are in these collections, which also means there’s no way to iterate over them.

String Iterators

Beginning with ECMAScript 5, JavaScript strings have slowly been evolving to be more array-like. For example, ECMAScript 5 formalizes bracket notation for accessing characters in strings (`text[0]` to get the first character). Unfortunately, bracket notation works on code units rather than characters, so it cannot be used to access double-byte characters correctly. ECMAScript 6 has added a lot of functionality to fully support Unicode (see Chapter 2) and as such, the default iterator for strings works on characters rather than code units.

Using bracket notation and the `length` property, the code units are used instead of characters and the output is a bit unexpected:

```
var message = "A ð ®· B";

for (let i=0; i < message.length; i++) {
    console.log(message[i]);
}
```

This code outputs the following:

```
A
(blank)
(blank)
(blank)
(blank)
B
```

Since the double-byte character is treated as two separate code units, there are four empty lines between `A` and `B` in the output.

Using the default string iterator with a `for-of` loop results in a more appropriate result:

```
var message = "A ð ®· B";

for (let c of message) {
    console.log(c);
}
```

This code outputs the following:

```
A
(blank)
ð ®·
(blank)
B
```

This output is more in line with what you might expect when working with characters. The default string iterator is ECMAScript 6's attempt at solving the iteration problem by using characters instead of code units.

NodeList Iterators

In the Document Object Model (DOM), there is a `NodeList` type that represents a collection of elements in a document. For those who write JavaScript to run in web browsers, understanding the difference between `NodeList` objects and arrays has always been a bit difficult. Both use the `length` property to indicate the number of items and both use bracket notation to access individual items. However, internally a `NodeList` and an array behave quite differently, and so that has led to a lot of confusion.

With the addition of default iterators in ECMAScript 6, the DOM definition of `NodeList` now specifically includes a default iterator that behaves in the same manner as the array default iterator. That means you can use NodeList in a for-of loop or any other place that uses an object's default iterator. For example:

```
var divs = document.getElementsByTagName("div");

for (let div of divs) {
    console.log(div.id);
}
```

This code uses getElementsByTagName() method to retrieve a NodeList that represents all of the <div> elements in the document. The for-of loop then iterates over each element and outputs its ID, effectively making the code the same as it would be for a standard array.

The Spread Operator

In Chapter 7, you saw how the spread operator (...) can be used to convert a set into an array. For example:

```
let set = new Set([1, 2, 3, 3, 3, 4, 5]),
    array = [...set];

console.log(array);             // [1,2,3,4,5]
```

This code uses the spread operator inside of an array literal to fill in an array with the values from set. The spread operator works on all iterables and uses the default iterator to determine which values to include. All values are read from the iterator and then inserted into the array in the location of the spread operator and in the order in which values where returned from the iterator. This example works because sets are iterables, but it can work equally well on any iterable. Here's another example:

```
let map = new Map([ ["name", "Nicholas"], ["age", 25]]),
    array = [...map];

console.log(array);         // [ ["name", "Nicholas"], ["age", 25]]
```

Here, map is converted into an array of arrays by using the spread operator. Since the default iterator for maps returns key-value pairs, the resulting array looks like the array that was passed into new Map().

You're not limited to the number of times you can use the spread operator in an array literal. You can use it wherever you want to insert multiple items from an iterable. For example:

```
let smallNumbers = [1, 2, 3],
    bigNumbers = [100, 101, 102],
    allNumbers = [0, ...smallNumbers, ...bigNumbers];

console.log(allNumbers.length);     // 7
console.log(allNumbers);    // [0, 1, 2, 3, 100, 101, 102]
```

Here, the spread operator is used to create allNumbers from the values in smallNumbers and bigNumbers. The values end up in the order in which the arrays appear in the array literal, so zero is first, followed by the values from smallNumbers, followed by the values from bigNumbers. Keep in mind that the original arrays are unchanged, only the values from those arrays have been copied into allNumbers.

Since the spread operator can be used on any iterable, it's the easiest way to convert an iterable into an array. That means you can convert strings into an array of characters (not code units) and NodeList objects in the browser in an array of nodes.

Now that you understand the basics of how iterators work, including for-of and the spread operator, it's time to look at some of the more complex uses of iterators.

Advanced Functionality

There's a lot that can be accomplished with the basic functionality of iterators and the convenience of creating them using generators. However, developers have discovered that iterators are much more powerful when used for tasks other than simply iterating over a collection of values. During the development of ECMAScript 6, a lot of unique ideas and patterns emerged that caused the addition of more functionality. Some of the changes are subtle, but when used together, can accomplish some interesting interactions.

Passing Arguments to Iterators

Throughout this chapter, you've seen that iterators can pass values out via the next() method or by using yield in a generator. It's also possible to pass arguments into the iterator through the next() method. When an argument is passed to next(), it becomes the value of the yield statement inside a generator. For example:

```
function *createIterator() {
    let first = yield 1;
    let second = yield first + 2;       // 4 + 2
    yield second + 3;                   // 5 + 3
}

let iterator = createIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next(4));          // "{ value: 6, done: false }"
console.log(iterator.next(5));          // "{ value: 8, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

The first call to next() is a special case where any argument passed to it is lost. Since arguments passed to next() become the value returned by yield, there would have to be a way to access that argument before the first yield in the generator function. That's not possible, so there's no reason to pass an argument the first time next() is called.

On the second call to next(), the value 4 is passed as the argument. The 4 ends up assigned to the variable first inside the generator function. In a yield statement including an assignment the right side of the expression is evaluated on the first call to next() and the left side is evaluated on the second call to next() before the function continues executing. Since the second call to next() passes in 4, that value is assigned to first and then execution continues.

The second yield uses the result of the first yield and adds two, which means it returns a value of six. When next() is called a third time, the value 5 is passed as an argument. That value is assigned to the variable second and then used in the third yield statement to return eight.

It's a bit easier to think about what's happening by considering which code is executing each time execution continues inside the generator function. Figure 6-1 uses colors to show the code being executed before yielding.

Figure 6-1: Code execution inside a generator

The color yellow represents the first call to next() and all of the code that is executed inside of the generator as a result; the color aqua represents the call to next(4) and the code that is executed; the color purple represents the call to next(5) and the code that is executed as a result. The tricky part is the code on the right side of each expression executing and stopping before the left side is executed. This makes debugging complicated generators a bit more involved than regular functions.

So far, you've seen that yield can act like return when a value is passed to next(). However, that's not the only execution trick inside of generators. You can also cause iterators throw an error.

Throwing Errors in Iterators

It's not only possible to pass data into iterators, it's also possible to pass error conditions. Iterators can choose to implement a throw() method that instructs the iterator to throw an error when it resumes. You can pass in an error object that should be thrown when the iterator continues processing. For example:

```
function *createIterator() {
    let first = yield 1;
    let second = yield first + 2;       // yield 4 + 2, then throw
    yield second + 3;                  // never is executed
}

let iterator = createIterator();

console.log(iterator.next());                   // "{ value: 1, done: false }"
console.log(iterator.next(4));                  // "{ value: 6, done: false }"
console.log(iterator.throw(new Error("Boom"))); // error thrown from generator
```

In this example, the first two yield expressions are evaluated as normal, but when throw() is called, an error is thrown before let second is evaluated. This effectively halts code execution similar to directly throwing an error. The only difference is the location in which the error is thrown. Figure 6-2 shows which code is executed at each step.

Figure 6-2: Throwing an error inside a generator

In this figure, the color red represents the code executed when throw() is called and the red star shows approximately when the error is thrown inside the generator. The first two yield statements are executed, it's only when throw() is called that an error is thrown before any other code is executed. Knowing this, it's possible to catch such errors inside the generator using a try-catch block, such as:

```
function *createIterator() {
    let first = yield 1;
    let second;

    try {
        second = yield first + 2;       // yield 4 + 2, then throw
    } catch (ex) {
        second = 6;                     // on error, assign a different value
    }
    yield second + 3;
}

let iterator = createIterator();

console.log(iterator.next());                   // "{ value: 1, done: false }"
console.log(iterator.next(4));                  // "{ value: 6, done: false }"
console.log(iterator.throw(new Error("Boom"))); // "{ value: 9, done: false }"
console.log(iterator.next());            // "{ value: undefined, done: true }"
```

In this example, a `try-catch` block is wrapped around the second `yield` statement. While this `yield` executes without error, the error is thrown before any value can be assigned to `second`, so the `catch` block assigns it a value of six. Execution then flows to the next `yield` and returns nine.

You'll also notice something interesting happened - the `throw()` method returned a value similar to that returned by `next()`. Because the error was caught inside the generator, code execution continued on to the next `yield` and returned the appropriate value.

It helps to think of `next()` and `throw()` as both being instructions to the iterator: `next()` instructs the iterator to continue executing (possibly with a given value) and `throw()` instructs the iterator to continue executing by throwing an error. What happens after that point depends on the code inside the generator.

The `next()` and `throw()` methods control execution inside of an iterator when using `yield`, but you can also use the `return` statement. However, it works a bit differently than in regular functions.

Generator Return Statements

Since generators are functions, you can use the `return` statement both to exit early and to specify a return value for the last call to `next()`. For most of this chapter you've seen examples where the last call to next() on an iterator returns undefined. It's possible to specify an alternate value by using return as you would in any other function. In a generator, return indicates that all processing is done, so the done property is set to true and the value, if provided, becomes the value field. Here's an example that simply exits early using return:

```
function *createIterator() {
    yield 1;
    return;
    yield 2;
    yield 3;
}

let iterator = createIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

In this code, the generator has a yield statement followed by a return statement. The return indicates that there are no more values to come and so the rest of the yield statements will not execute (they are unreachable).

You can also specify a return value that will end up in the value field of the returned object. For example:

```
function *createIterator() {
    yield 1;
    return 42;
}

let iterator = createIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 42, done: true }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

Here, the value 42 is returned in the value field on the second call to next() (which is the first time that done is true). The third call to next() returns an object whose value property is once again undefined. Any value you specify with return is only available on the returned object one time before the value field is reset to undefined.

Any value specified by return is ignored by for-of and the spread operator. As soon as they see done is true, they stop without reading the value.

Delegating Generators

In some cases it may be useful to combine the values from two iterators into one. Using generators, it's possible to delegate to another generator using a special form of yield with a star (*). As with generator definitions, it doesn't matter where the star appears so as long as it is between the keyword yield and the generator function name. Here's an example:

```
function *createNumberIterator() {
    yield 1;
    yield 2;
}

function *createColorIterator() {
    yield "red";
    yield "green";
}

function *createCombinedIterator() {
    yield *createNumberIterator();
    yield *createColorIterator();
    yield true;
}

var iterator = createCombinedIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: "red", done: false }"
console.log(iterator.next());           // "{ value: "green", done: false }"
console.log(iterator.next());           // "{ value: true, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

In this example, the createCombinedIterator() generator delegates first to createNumberIterator() and then to createColorIterator(). The returned iterator appears, from the outside, to be one consistent iterator that has produced all of the values.

Each call to next() is delegated to the appropriate iterator until they are empty, and then the final yield is executed to return true.

Generator delegation also lets you make use of generator return values (as seen in the previous section). This is the easiest way to access such returned values and can be quite useful in performing complex tasks. For example:

```
function *createNumberIterator() {
    yield 1;
    yield 2;
    return 3;
}

function *createRepeatingIterator(count) {
    for (let i=0; i < count; i++) {
        yield "repeat";
    }
}

function *createCombinedIterator() {
    let result = yield *createNumberIterator();
    yield *createRepeatingIterator(result);
}

var iterator = createCombinedIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

Here, the createCombinedIterator() generator delegates to createNumberIterator() and assigns the return value to result. Since createNumberIterator() contains return 3, the returned value is 3. The result variable is then passed to createRepeatingIterator() as an argument indicating how many times to yield the same string (in this case, three times).

Notice that the value 3 was never output from any call to next(), it existed solely inside of createCombinedIterator(). It is possible to output that value as well by adding another yield statement, such as:

```
function *createNumberIterator() {
    yield 1;
    yield 2;
    return 3;
}

function *createRepeatingIterator(count) {
    for (let i=0; i < count; i++) {
        yield "repeat";
    }
}

function *createCombinedIterator() {
    let result = yield *createNumberIterator();
    yield result;
    yield *createRepeatingIterator(result);
}

var iterator = createCombinedIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

In this code, the extra `yield` statement explicitly outputs the returned value from `createNumberIterator()`.

Generator delegation using the return value is a very powerful paradigm that allows for some very interesting possibilities, especially when used in conjunction with asynchronous operations.

> You can use `yield *` directly on strings, such as `yield * "hello"` and the string’s default iterator will be used.

Asynchronous Task Running

A lot of the excitement around generators is directly related to usage with asynchronous programming. Asynchronous programming in JavaScript is a double-edged sword: it's very easy to do simple things while complex things become an errand in code organization. Since generators allow you to effectively pause code in the middle of execution, this opens up a lot of possibilities as it relates to asynchronous processing.

The traditional way to perform asynchronous operations is to call a function that has a callback. For example, consider reading a file from disk in Node.js:

```
let fs = require("fs");

fs.readFile("config.json", function(err, contents) {
    if (err) {
        throw err;
    }

    doSomethingWith(contents);
    console.log("Done");
});
```

The `fs.readFile()` method is called with the filename to read and a callback function. When the operation is finished, the callback function is called. The callback checks to see if there's an error, and if not, processes the returned `contents`. This works well when you have a small, finite number of asynchronous tasks to complete, but gets complicated when you need to nest callbacks or otherwise sequence a series of asynchronous tasks. This is where generators and `yield` are helpful.

A Simple Task Runner

Because `yield` stops execution and waits for the `next()` method to be called before starting again, this provides a way to implement asynchronous calls without managing callbacks. To start, you need a function that can call a generator and start the iterator, such as:

```
function run(taskDef) {

    // create the iterator, make available elsewhere
    let task = taskDef();

    // start the task
    let result = task.next();

    // recursive function to keep calling next()
    function step() {

        // if there's more to do
        if (!result.done) {
            result = task.next();
            step();
        }
    }

    // start the process
    step();

}
```

The run() function accepts a task definition (a generator function) as an argument. It calls the generator to create an iterator and stores the iterator in task. The task variable is outside of the function so it can be accessed by other functions (the reason why will be clear later in this section). The first call to next() begins the iterator and the result is stored for later use. The function step() checks to see if result.done is false and, if so, calls next() before recursively calling itself. Each call to next() stores the return value in result, so that variable is always overwritten to contain the latest information. The initial call to step() starts the process of looking at result.done.

With this implementation of run(), you can run a generator containing multiple yield statements, such as:

```
run(function*() {
    console.log(1);
    yield;
    console.log(2);
    yield;
    console.log(3);
});
```

This example just outputs three numbers to the console, which simply shows that all calls to `next()` are being made. However, just yielding a couple times of times isn't very useful, so the next step is to pass values into and out of the iterator.

Task Running With Data

The easiest way to pass data through the task runner is to pass the value specified by `yield` into the next call to `next()`. To do so, you need only pass `result.value`, as in this code:

```
function run(taskDef) {

    // create the iterator, make available elsewhere
    let task = taskDef();

    // start the task
    let result = task.next();

    // recursive function to keep calling next()
    function step() {

        // if there's more to do
        if (!result.done) {
            result = task.next(result.value);
            step();
        }
    }

    // start the process
    step();

}
```

With this change, it's now possible to pass data back and forth, such as:

```
run(function*() {
    let value = yield 1;
    console.log(value);         // 1

    value = yield value + 3;
    console.log(value);         // 4
});
```

This example outputs two values to the console: 1 and 4. The value 1 comes from yield 1, as the 1 is being passed right back into the variable value. The 4 is calculated by adding 3 to value and passing that result back to value. Now that data is flowing between calls to yield, it's just a small change to allow asynchronous calls.

Asynchronous Task Runner

Whereas the previous example passed static data back and forth, waiting for an asynchronous process is slightly different. The task runner needs to know about callbacks and how to use them. And since yield expressions pass their values into the task runner, that means any function call must return a value that somehow indicates it's an asynchronous operation that the task runner should wait for. How can you signal that?

For the purposes of this example, any function meant to be called by the task runner will return a function that executes a callback. For example:

```
function fetchData() {
    return function(callback) {
        callback(null, "Hi!");
    };
}
```

The fetchData() function returns a function that accepts a callback function as an argument. When the returned function is called, it executes the callback function with a single piece of data, the string "Hi!". The callback argument needs to come from the task runner and ensure that calling it correctly interacts with the underlying iterator. While the fetchData() function is synchronous, you can easily extend it to be asynchronous by calling the callback with a slight delay, such as:

```
function fetchData() {
    return function(callback) {
        setTimeout(function() {
            callback(null, "Hi!");
        }, 50);
    };
}
```

This version of fetchData() introduces a 50ms delay before calling the callback, demonstrating that this pattern works equally well for synchronous and asynchronous code. You just have to make sure each function that wants to be called using yield follows this same pattern.

With a good understanding of how functions signal that they are an asynchronous process, you can modify the task runner to take this into account. Anytime result.value is a function, the task runner will execute it instead of just passing that value into next(). Here’s the updated code:

```
function run(taskDef) {

    // create the iterator, make available elsewhere
    let task = taskDef();

    // start the task
    let result = task.next();

    // recursive function to keep calling next()
    function step() {

        // if there's more to do
        if (!result.done) {
            if (typeof result.value === "function") {
                result.value(function(err, data) {
                    if (err) {
                        result = task.throw(err);
                        return;
                    }

                    result = task.next(data);
                    step();
                });
            } else {
                result = task.next(result.value);
                step();
            }

        }
    }

    // start the process
    step();

}
```

When `result.value` is a function, it is called with a callback function. That callback function follows the Node.js convention of passing any possible error as the first argument and the result as the second argument. If `err` is present, then that means an error occurred and `task.throw()` is called with the error object instead of `task.next()` so an error is thrown at the correct location. If there is no error, then `data` is passed into `task.next()` and the result is stored. Then, `step()` is called to continue the process. When result.value is not a function, it is directly passed into next() just as before.

This new version of the task runner is ready for all asynchronous tasks. To read data from a file in Node.js, you need to create a wrapper around fs.readFile() that returns a function similar to the fetchData() function from earlier in this section. For example:

```
let fs = require("fs");

function readFile(filename) {
    return function(callback) {
        fs.readFile(filename, callback);
    };
}
```

The readFile() method accepts a single argument, the filename, and returns a function that calls a callback. The callback is passed directly to fs.readFile(), which will execute it upon completion. You can then run this task using yield as follows:

```
run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

This example is performing the asynchronous readFile() operation without making any callbacks visible in the main code. Aside from yield, the code looks the same as synchronous code. As long as the functions performing asynchronous operations all conform to the same interface, you can write logic that reads like synchronous code.

Of course, there are downsides to the pattern used in these examples, namely that you can’t always be sure that a function that returns a function is asynchronous. For now, though, it’s only important that you understand the theory behind the task running. There are more powerful ways of doing asynchronous task scheduling using promises, and that will be covered further in Chapter 11.

Summary

Iterators are an important part of ECMAScript 6 and are at the root of several important parts of the language. On the surface, iterators provide a simple way to return a sequence of values using a simple API. However, there are far more complex ways to use iterators in ECMAScript 6.

The @@iterator symbol is used to define default iterators for objects. Both built-in objects and developer-defined objects can use this symbol to provide a method that returns an iterator. When @@iterator is provided, the object is considered an iterable.

The for-of loop uses iterables to return a series of values in a loop. This makes creating loops easier than the traditional for loop because you no longer need to track values and control when the loop ends. The for-of loop automatically reads all values from the iterator until there are no more and then exits.

To make it easier to use for-of, many values in ECMAScript 6 have default iterators. All the collection types, arrays, maps, and sets, have iterators designed for easy access to their contents. Strings also have a default iterator so it's easy to iterate over the characters of the string (rather than the code units).

The spread operator works with any iterable and makes it easy to convert iterables into arrays. It does this by reading values from an iterator and inserting them individually into an array.

Generators are a special type of function that automatically creates an iterator when called. These functions are indicated by the start (*) and make use of the yield keyword to indicate which value to return for each successive call to next().

Generator delegation encourages good encapsulation of iterator behavior by letting you reuse existing generators in new ones. This is done using yield * instead of yield, allowing you to create an iterator that returns values from multiple iterators.

Perhaps the most interesting and exciting aspect of generators and iterators is the possibility of creating cleaner-looking asynchronous code. Instead of needing to use callbacks everywhere, you can setup code that looks synchronous but in fact uses yield to wait for asynchronous operations to complete.

Classes

Ever since JavaScript was first created, many developers have been confused by its lack of classes. Most formal object-oriented programming languages support classes and classical inheritance as the primary way of defining similar and related objects. From pre-ECMAScript 1 all the way through ECMAScript 5, this point of confusion led many libraries to create utilities designed to make JavaScript look like it had support for classes.

While there are some JavaScript developers who feel strongly that the language doesn't need classes, the fact that so many libraries were created specifically for this purpose led to the inclusion of classes in ECMAScript 6. However, ECMAScript 6 classes aren't exactly the same as classes in other languages. There's a uniqueness about them that embraces the dynamic nature of JavaScript.

Class-Like Structures in ECMAScript 5

Before exploring classes, it's helpful to understand the underlying mechanisms that classes use. In ECMAScript 5 and earlier, there were no classes, and the closest equivalent was creating a constructor and then assigning methods to its prototype. This approach is called creating a custom type. For example:

```
function PersonType(name) {
    this.name = name;
}

PersonType.prototype.sayName = function() {
    console.log(this.name);
};

let person = new PersonType("Nicholas");
person.sayName();   // outputs "Nicholas"

console.log(person instanceof PersonType);  // true
console.log(person instanceof Object);      // true
```

In this code, `PersonType` is a constructor function that creates a single property called `name`. The `sayName()` method is assigned to the prototype so the same function is shared by all instances of `PersonType`. Then, a new instance of `PersonType` is created via the `new` operator, and the resulting `person` object is considered an instance of `PersonType` and of `Object` (through prototypal inheritance).

This same basic pattern underlies a lot of the class-mimicking JavaScript libraries. And that's where ECMAScript 6 classes start.

Class Declarations

The simplest class form is the one that looks similar to other languages: the class declaration. Class declarations begin with the class keyword followed by the name of the class. The rest of the syntax looks similar to concise methods in object literals without requiring commas between them. For example, here's the class equivalent of the previous example:

```
class PersonClass {

    // equivalent of the PersonType constructor
    constructor(name) {
        this.name = name;
    }

    // equivalent of PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
}

let person = new PersonClass("Nicholas");
person.sayName();   // outputs "Nicholas"

console.log(person instanceof PersonClass);     // true
console.log(person instanceof Object);          // true

console.log(typeof PersonClass);                    // "function"
console.log(typeof PersonClass.prototype.sayName);  // "function"
```

The class declaration PersonClass behaves quite similarly to PersonType from the previous example. Instead of defining a function as the constructor, class declarations allow you to define the constructor directly inside of the class using the special constructor method name. Since class methods use the concise syntax, there's no need to use the function keyword. All other method names have no special meaning, so you can add as many as you want.

Own properties, properties that occur on the instance rather than the prototype, can only be created inside of a class constructor or method. In the previous example, `name` is an own property. It’s recommended to create all possible own properties inside of the constructor function so there’s a single place that’s responsible for all of them.

An interesting aspect of class declarations is that they are just syntactic sugar on top of the existing custom type declarations. The `PersonClass` declaration actually creates a function that has the behavior of the `constructor` method, which is why `typeof PersonClass` is `"function"`. Similarly, the `sayName()` method ends up as a method on `PersonClass.prototype`, similar to `PersonType.prototype` in the earlier example. These similarities allow you to mix custom types and classes without worrying too much about which you’re using.

Despite the similarities, there are some important differences to keep in mind:

1. Class declarations, unlike function declarations, are not hoisted. Class declarations act like `let` declarations and so exist in the temporal dead zone until execution reaches the declaration.
2. All code inside of class declarations runs in strict mode automatically. There’s no way to opt-out of strict mode inside of classes.
3. All methods are non-enumerable. This is a significant change from custom types, where you need to use `Object.defineProperty()` to make a method non-enumerable.
4. All methods have no `[[Construct]]` internal method and so throw an error if you try to call them with `new`.
5. Calling the class constructor without `new` throws an error.
6. Attempting to overwrite the class name within a class method throws an error.

With all of this in mind, the `PersonClass` declaration from the previous example is directly equivalent to the following:

```
// direct equivalent of PersonClass
let PersonType2 = (function() {

    "use strict";

    const PersonType2 = function(name) {

        // make sure the function was called with new
        if (typeof new.target === "undefined") {
            throw new Error("Constructor must be called with new.");
        }

        this.name = name;
    }

    Object.defineProperty(PersonType2.prototype, "sayName", {
        value: function() {

            // make sure the method wasn't called with new
            if (typeof new.target !== "undefined") {
                throw new Error("Method cannot be called with new.");
            }

            console.log(this.name);
        },
        enumerable: false,
        writable: true,
        configurable: true
    });

    return PersonType2;
}());
```

The first thing to notice in this code is that there are two `PersonType2` declarations, a `let` declaration in the outer scope and a `const` declaration inside of the IIFE. This is how class methods are forbidden from overwriting the class name while code outside of the class is allowed to do so. The constructor function checks `new.target` to ensure that it’s being called with `new`, otherwise an error is thrown. Next, the `sayName()` method is defined as nonenumerable and it also checks `new.target` to ensure that it wasn’t called with `new`. The final step is to return the constructor function.

From this example, you can see that while it is possible to do everything that classes do without using new syntax, the class syntax makes all of the functionality a lot simpler than it would be otherwise.

Constant Class Names

The name of a class is specified as if using const, but only inside of the class itself. That means you can overwrite the class name outside of the class but not inside a class method. For example:

```
class Foo {
  constructor() {
     Foo = "bar";    // throws an error when executed
  }
}

// but this is okay
Foo = "baz";
```

In this code, the Foo inside of the class constructor is a separate binding than the Foo outside of the class. The internal Foo is defined as if it's a const and so cannot be overwritten, which means an error is thrown when an attempt is made to do so; the external Foo is defined as if it's a let declaration and so its value may be overwritten at any time.

Class Expressions

Classes and functions are similar in that they have two forms: declarations and expressions. Function and class declarations begin with an appropriate keyword (function or class, respectively) followed by an identifier. Functions have an expression form that doesn't require an identifier after function, and similarly, classes have an expression form that doesn't require an identifier after class.

These class expressions are designed to be used in variable declarations or passed into functions as arguments. Here's the class expression equivalent of the previous examples:

```
// class expressions do not require identifiers after "class"
let PersonClass = class {

    // equivalent of the PersonType constructor
    constructor(name) {
        this.name = name;
    }

    // equivalent of PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
};

let person = new PersonClass("Nicholas");
person.sayName();   // outputs "Nicholas"

console.log(person instanceof PersonClass);     // true
console.log(person instanceof Object);          // true

console.log(typeof PersonClass);                    // "function"
console.log(typeof PersonClass.prototype.sayName);  // "function"
```

Aside from the syntax, class expressions are exactly equivalent to class declarations.

Whether you use class declarations or class expressions is purely a matter of style. Unlike function declarations and function expressions, both class declarations and class expressions are not hoisted, and so the choice has little bearing on the runtime behavior of the code.

Named Class Expressions

The previous section used an anonymous class expression in the example, but you can also name class expressions just like you can name function expressions. To do so, include an identifier after the `class` keyword:

```
let PersonClass = class PersonClass2 {

    // equivalent of the PersonType constructor
    constructor(name) {
        this.name = name;
    }

    // equivalent of PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
};

console.log(typeof PersonClass);        // "function"
console.log(typeof PersonClass2);       // "undefined"
```

In this example, the class expression is given a name, PersonClass2. The PersonClass2 identifier exists only within the class definition so that it can be used inside of methods. Outside of the class, there is no PersonClass2 binding, so typeof PersonClass2 is "undefined". To understand why this is, here's the equivalent declaration without using classes:

```
// direct equivalent of PersonClass named class expression
let PersonClass = (function() {

    "use strict";

    const PersonClass2 = function(name) {

        // make sure the function was called with new
        if (typeof new.target === "undefined") {
            throw new Error("Constructor must be called with new.");
        }

        this.name = name;
    }

    Object.defineProperty(PersonClass2.prototype, "sayName", {
        value: function() {
            console.log(this.name);
        },
        enumerable: false,
        writable: true,
        configurable: true
    });

    return PersonClass2;
}());
```

Creating a named class expression slightly changes what's happening in the JavaScript engine. For class declarations the outer binding (defined with `let`) has the same name as the inner binding (defined with `const`). Named class expressions use their name as the `const` definition, so in this case, `PersonClass2` is defined for use only inside of the class.

While the behavior of named class expressions is different from that of named function expressions, there are still a lot of similarities between the two. Both can be used as values, and that opens up a lot of possibilities.

Classes as First-Class Citizens

In programming, something is said to be a first-class citizen when it can be used as a value, meaning it can be passed into a function, returned from a function, and assigned to a variable. JavaScript functions are first class citizens, sometimes just called first class functions, and that's part of what makes

JavaScript unique. ECMAScript 6 continues this tradition by making classes first-class citizens as well. That allows classes to be used in a lot of different ways. For example, they can be passed into functions as arguments:

```
function createObject(classDef) {
    return new classDef();
}

let obj = createObject(class {

    sayHi() {
        console.log("Hi!");
    }
});

obj.sayHi();        // "Hi!"
```

In this example, an anonymous class expression is passed into `createObject()`. An instance is then created by using `new` and that object is returned.

Another interesting use of class expressions is to create singletons by immediately invoking the class constructor. To do so, you must use `new` with a class expression and include parentheses at the end. For example:

```
let person = new class PersonClass {

    constructor(name) {
        this.name = name;
    }

    sayName() {
        console.log(this.name);
    }

}("Nicholas");

person.sayName();       // "Nicholas"
```

Here, the anonymous class expression is created and then executed immediately. This pattern allows you to use the class syntax for creating singletons without leaving a class reference available for inspection (remember that `PersonClass` only creates a binding inside of the class, not outside). The parentheses at the end are the indicator that you're calling a function while also allowing you to pass in an argument.

The examples in this chapter so far have focused on classes with methods. But you can also create accessor properties on classes using a syntax similar to object literals.

Accessor Properties

While own properties should be created inside of class constructors, classes allow you to define accessor properties on the prototype. To create a getter, use the keyword get followed by a space followed by an identifier; to create a setter, do the same using the keyword set. For example:

```
class CustomHTMLElement {

    constructor(element) {
        this.element = element;
    }

    get html() {
        return this.element.innerHTML;
    }

    set html(value) {
        this.element.innerHTML = value;
    }
}

var descriptor = Object.getOwnPropertyDescriptor(CustomHTMLElement.prototype,\
 "html");
console.log("get" in descriptor);   // true
console.log("set" in descriptor);   // true
console.log(descriptor.enumerable); // false
```

In this example, the CustomHTMLElement class is made as a wrapper around an existing DOM element. It has both a getter and setter for html that delegates to the innerHTML method on the element itself. This accessor property is created as non-enumerable, just like any other method would be, and is created on the CustomHTMLElement.prototype. The equivalent non-class representation is:

```
// direct equivalent to previous example
let CustomHTMLElement = (function() {

    "use strict";

    const CustomHTMLElement = function(element) {

        // make sure the function was called with new
        if (typeof new.target === "undefined") {
            throw new Error("Constructor must be called with new.");
        }

        this.element = element;
    }

    Object.defineProperty(CustomHTMLElement.prototype, "html", {
        enumerable: false,
        configurable: true,
        get: function() {
            return this.element.innerHTML;
        },
        set: function(value) {
            this.element.innerHTML = value;
        }
    });

    return CustomHTMLElement;
}());
```

As with previous examples, this one shows just how much code you're saving by using a class instead of the non-class equivalent. The accessor property definition alone is almost the size of the equivalent class declaration.

The similarities between objects literals and classes aren't quite over yet. Class methods and accessor properties can use computed names using the same syntax as object literals.

Computed Member Names

Class methods and accessor properties can have computed names. Instead of using an identifier, use square brackets around an expression (the same as with object literal computed names). For example:

```
let methodName = "sayName";

class PersonClass {

    constructor(name) {
        this.name = name;
    }

    [methodName]() {
        console.log(this.name);
    }
};

let me = new PersonClass("Nicholas");
me.sayName();           // "Nicholas"
```

This version of PersonClass uses a variable to assign the method name. The methodName variable contains the string "sayName", which is then used to declare the method. The sayName() method is later accessed directly. Accessor properties can also use computed names in the same way:

```
let propertyName = "html";

class CustomHTMLElement {

    constructor(element) {
        this.element = element;
    }

    get [propertyName]() {
        return this.element.innerHTML;
    }

    set [propertyName](value) {
        this.element.innerHTML = value;
    }
}
```

Here, the getter and setter for html are set using the variable propertyName. You can still access the property by using .html, it's just the definition that is affected.

You've seen that there are a lot of similarities between classes and object literals, with methods, accessor properties, and computed names. There's just one more similarity to cover: generators.

Generator Methods

When generators were introduced in Chapter 8, you learned how to define a generator on an object literal by prepending a star (*) to the method name. The same syntax works for classes as well, allowing any method to be a generator. Here's an example:

```
class MyClass {

    *createIterator() {
        yield 1;
        yield 2;
        yield 3;
    }

}

let instance = new MyClass();
let iterator = instance.createIterator();
```

This code creates a class called `MyClass` that has a generator method called `createIterator()`. The method returns an iterator whose values are hardcoded into the generator. While this is a useful capability, it's much more useful to define a default iterator for your class.

You can define the default iterator for a class by using `Symbol.iterator` to define a generator method, such as:

```
class Collection {

    constructor() {
        this.items = [];
    }

    *[Symbol.iterator]() {
        yield *this.items.values();
    }
}

var collection = new Collection();
collection.items.push(1);
collection.items.push(2);
collection.items.push(3);

for (let x of collection) {
    console.log(x);
}

// Output:
// 1
// 2
// 3
```

This example uses a computed name for a generator method that delegates to the `values()` iterator of `this.items`. In this way, any instance of `Collection` can be used directly in a `for-of` loop or with the spread operator. It's a good idea to define a default iterator for any class that manages a collection of values.

Adding methods and accessor properties to a class prototype is useful when you want those to show up on object instances. If, on the other hand, you'd like methods or accessor properties on the class itself, then you'll need to use static members.

Static Members

Another common pattern in JavaScript is adding additional methods directly onto constructors to simulate static members. For example:

```
function PersonType(name) {
    this.name = name;
}

// static method
PersonType.create = function(name) {
    return new PersonType(name);
};

// instance method
PersonType.prototype.sayName = function() {
    console.log(this.name);
};

var person = PersonType.create("Nicholas");
```

This code creates a factory method called `PersonType.create()`. In other programming languages, this would be considered a static method as it is not dependent on an instance of `PersonType` for its data.

Classes simplify the creation of static members by using the formal `static` annotation before the method or accessor property name. Here’s the equivalent of the last example:

```
class PersonClass {

    // equivalent of the PersonType constructor
    constructor(name) {
        this.name = name;
    }

    // equivalent of PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }

    // equivalent of PersonType.create
    static create(name) {
        return new PersonClass(name);
    }
}

let person = PersonClass.create("Nicholas");
```

The `PersonClass` definition has a single static method called `create()`. The method syntax is the same as for `sayName()` with the exception of the `static` keyword. You can use the `static` keyword on any method or accessor property definition within a class. The only restriction is that you cannot use `static` with the `constructor` method definition.

⚠ Static members are not accessible from instances. You must always access static members from the class directly.

Inheritance with Derived Classes

Another problem with custom types in ECMAScript 5 and earlier was the extensive process necessary to implement inheritance. To properly inherit, you would need multiple steps. For instance:

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

function Square(length) {
    Rectangle.call(this, length, length);
}

Square.prototype = Object.create(Rectangle.prototype, {
    constructor: {
        value:Square,
        enumerable: true,
        writable: true,
        configurable: true
    }
});

var square = new Square(3);

console.log(square.getArea());              // 9
console.log(square instanceof Square);      // true
console.log(square instanceof Rectangle);   // true
```

Here, `Square` inherits from `Rectangle`, and to do so, it must overwrite `Square.prototype` with a new object created from `Rectangle.prototype` as well as call `Rectangle.call()`. These steps often confused newcomers to the language and were a source of errors for experienced developers.

Classes make inheritance easier by using the familiar `extends` keyword to specify the function from which the class should inherit. The prototypes are automatically adjusted and you can access the base class constructor using `super()`. Here's the equivalent of the previous example:

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}

class Square extends Rectangle {
    constructor(length) {

        // same as Rectangle.call(this, length, length)
        super(length, length);
    }
}

var square = new Square(3);

console.log(square.getArea());              // 9
console.log(square instanceof Square);      // true
console.log(square instanceof Rectangle);   // true
```

In this example, the `Square` class inherits from `Rectangle` using the `extends` keyword. The `Square` constructor uses `super()` to call the `Rectangle` constructor with the specified arguments. Note that unlike the ECMAScript 5 version of the code, the identifier `Rectangle` is only used within the class declaration (after `extends`).

Using `super()` is a requirement of derived classes if you specify a constructor (if you don't, an error will occur). If you choose not to use a constructor, then `super()` is automatically called for you with all arguments upon creating a new instance of the class. For instance, the following two classes are identical:

```
class Square extends Rectangle {
    // no constructor
}

// Is equivalent to

class Square extends Rectangle {
    constructor(...args) {
        super(...args);
    }
}
```

The second class in this example shows the equivalent of the default constructor for all derived classes. All of the arguments are passed, in order, to the base class constructor. In this case, the functionality isn't quite correct because the `Square` constructor needs only one argument and so it's best to manually define the constructor.

There are a few things to keep in mind when using `super()`:

1. You can only use `super()` in a derived class. If you try to use it in a non-derived class (a class that doesn't use `extends`) or a function, it will throw an error.
2. You must call `super()` before accessing `this` in the constructor. Since `super()` is responsible for initializing `this`, attempting to access `this` before calling `super()` results in an error.
3. The only way to avoid calling `super()` is to return an object from the class constructor.

Shadowing Class Methods

The methods on derived classes always shadow methods of the same name on the base class. For instance, you can add `getArea()` to `Square` in order to redefine that functionality:

```
class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }

    // override and shadow Rectangle.prototype.getArea()
    getArea() {
        return this.length * this.length;
    }
}
```

In this code, getArea() is now defined as part of Square and therefore Rectangle.prototype.getArea() will no longer be called by any instances of Square. Of course, you can always decide to call the base class version of the method by using super.getArea(), such as:

```
class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }

    // override, shadow, and call Rectangle.prototype.getArea()
    getArea() {
        return super.getArea();
    }
}
```

Using super in this way is the same as discussed in Chapter 4: the this value is automatically set correctly so you can make a simple method call.

Inherited Static Members

If a base class has static members then those static members are also available on the derived class. This maps to how inheritance works in other languages, but is a new concept for JavaScript. Here's an example:

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }

    static create(length, width) {
        return new Rectangle(length, width);
    }
}

class Square extends Rectangle {
    constructor(length) {

        // same as Rectangle.call(this, length, length)
        super(length, length);
    }
}

var rect = Square.create(3, 4);

console.log(rect instanceof Rectangle);     // true
console.log(rect.getArea());                // 12
console.log(rect instanceof Square);        // false
```

In this code, a new static `create()` method is added to `Rectangle`. Through inheritance, that method is available as `Square.create()` and behaves in the same manner as `Rectangle.create()`.

Derived Classes from Expressions

Perhaps the most powerful aspect of derived classes in ECMAScript 6 is the ability to derive a class from an expression. You can use `extends` with any expression as long as the expression resolves to a function with `[[Construct]]` and a prototype. For example:

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());              // 9
console.log(x instanceof Rectangle);   // true
```

This example defines Rectangle as an ECMAScript 5-style constructor while Square is a class. Since Rectangle has [[Construct]] and a prototype, the class can still inherit directly from it.

Accepting any type of expression after extends allows for some powerful possibilities, such as dynamically determining what to inherit from. For example:

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

function getBase() {
    return Rectangle;
}

class Square extends getBase() {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());               // 9
console.log(x instanceof Rectangle);    // true
```

Here, the `getBase()` function is called directly as part of the class declaration. It returns `Rectangle`, which means this example is functionally equivalent to the previous one. And since you can determine the base dynamically, that means it's possible to create different inheritance approaches. For instance, you can effectively create mixins:

```
let SerializableMixin = {
    serialize() {
        return JSON.stringify(this);
    }
};

let AreaMixin = {
    getArea() {
        return this.length * this.width;
    }
};

function mixin(...mixins) {
    var base = function() {};
    Object.assign(base.prototype, ...mixins);
    return base;
}

class Square extends mixin(AreaMixin, SerializableMixin) {
    constructor(length) {
        super();
        this.length = length;
        this.width = length;
    }
}

var x = new Square(3);
console.log(x.getArea());               // 9
console.log(x.serialize());             // "{"length":3,"width":3}"
```

In this example, mixins are used instead of classical inheritance. The mixin() function takes any number of arguments that represent mixin objects. It creates a function called base and assigns the properties of each mixin object to the prototype. The function is then returned so Square can use extends. Keep in mind that since extends is still used, you are required to call super() in the constructor.

The instance of Square has both getArea() from AreaMixin and serialize from SerializableMixin. This is accomplished through prototypal inheritance, as the mixin() function dynamically populates the prototype of a new function with all of the own properties of each mixin.

Even though any expression can be used after `extends`, not all expressions result in a valid class. Specifically, the following expression types causes errors:

- `null`
- generator functions (chapter 8)

In these cases, attempting to create a new instance of the class will throw an error because there is no `[[Construct]]` to call.

Inheriting from Built-ins

For almost as long as there have been JavaScript arrays, developers have wanted to inherit from arrays to create their own special array types. However, in ECMAScript 5 and earlier, this wasn't possible. Attempting to use classical inheritance didn't result in functioning code. For example:

```
// built-in array behavior
var colors = [];
colors[0] = "red";
console.log(colors.length);         // 1

colors.length = 0;
console.log(colors[0]);             // undefined

// trying to inherit from array in ES5

function MyArray() {
    Array.apply(this, arguments);
}

MyArray.prototype = Object.create(Array.prototype, {
    constructor: {
        value: MyArray,
        writable: true,
        configurable: true,
        enumerable: true
    }
});

var colors = new MyArray();
colors[0] = "red";
console.log(colors.length);         // 0

colors.length = 0;
console.log(colors[0]);             // "red"
```

As you can see, using the classical form of JavaScript inheritance results in unexpected behavior. The `length` and numeric properties don't behave the same as the built-in array because this functionality isn't covered either by `Array.apply()` or by assigning the prototype.

One of the goals of ECMAScript 6 classes is to allow inheritance from all built-ins. In order to accomplish this, the inheritance model of classes is slightly different than the classical inheritance model found in ECMAScript 5 and earlier:

- In ECMAScript 5 classical inheritance, the value of `this` is first created by the derived type (for example, `MyArray`) and then the base type constructor is called (`Array.apply()`). That means `this` starts out as an instance of `MyArray` and then is decorated with additional properties from Array.

- In ECMAScript 6 class-based inheritance, the value of this is first created by the base (Array) and then modified by the derived class constructor (MyArray). The result is that this starts out with all of the built-in functionality of the base and correctly receives all functionality related to it.

The following class-based special array works as you would expect:

```
class MyArray extends Array {
    // empty
}

var colors = new MyArray();
colors[0] = "red";
console.log(colors.length);         // 1

colors.length = 0;
console.log(colors[0]);             // undefined
```

In this example, MyArray inherits directly from Array and therefore works in the exact same way. Interacting with numeric properties updates the length property, and manipulating the length property updates the numeric properties. That means not only can you properly inherit from Array to create your own derived array classes, you can also inherit from other builtins as well. ECMAScript 6 and derived classes have effectively removed the last special case of inheriting from builtins.

The @@species Property

An interesting aspect of inheriting from builtins is that any method that returns an instance of the builtin will automatically return a derived class instance instead. So, if you have a derived class MyArray that inherits from Array, methods such as slice() return an instance of MyArray. For example:

```
class MyArray extends Array {
    // empty
}

let items = new MyArray(1, 2, 3, 4),
    subitems = items.slice(1, 3);

console.log(items instanceof MyArray);      // true
console.log(subitems instanceof MyArray);   // true
```

In this code, the slice() method returns a MyArray instance. The slice() method is inherited from Array and returns an instance of Array normally. Behind the scenes, it's the @@species property that is making this change.

The @@species well-known symbol is used to define a static accessor property that returns a function. That function is a constructor to use whenever an instance of the class must be created inside of an instance method (instead of using the constructor). There are several builtin types that have @@species defined:

- Array
- ArrayBuffer (discussed in Chapter 10)
- Map
- Promise
- RegExp
- Set
- Typed Arrays (discussed in Chapter 10)

Each of these types have a default @@species property that returns this, meaning that it will always return the constructor function. If you were to do the same on a custom class, the code would look like this:

```
// several builtin types use species similar to this
class MyClass {
    static get [Symbol.species]() {
        return this;
    }

    constructor(value) {
        this.value = value;
    }

    clone() {
        return new this.constructor[Symbol.species](this.value);
    }
}
```

In this example, the Symbol.species well-known symbol is used to assign a static accessor property to MyClass. Note that there's only a getter without a setter, because it is not possible to change the species of a class. Any call to this.constructor[Symbol.species] returns MyClass. The clone() method uses that definition to return a new instance rather than directly using MyClass, which allows derived classes to override that value. For example:

```
class MyClass {
    static get [Symbol.species]() {
        return this;
    }

    constructor(value) {
        this.value = value;
    }

    clone() {
        return new this.constructor[Symbol.species](this.value);
    }
}

class MyDerivedClass1 extends MyClass {
    // empty
}

class MyDerivedClass2 extends MyClass {
    static get [Symbol.species]() {
        return MyClass;
    }
}

let instance1 = new MyDerivedClass1("foo"),
    clone1 = instance1.clone(),
    instance2 = new MyDerivedClass2("bar"),
    clone2 = instance2.clone();

console.log(clone1 instanceof MyClass);             // true
console.log(clone1 instanceof MyDerivedClass1);     // true
console.log(clone2 instanceof MyClass);             // true
console.log(clone2 instanceof MyDerivedClass2);     // false
```

Here, DerivedClass1 inherits from MyClass and doesn't change the @@species property. When clone() is called, it returns an instance of MyDerivedClass1 because this.constructor[Symbol.species] returns MyDerivedClass1. The DerivedClass2 class inherits from MyClass and overrides @@species to return MyClass. When clone() is called on an instance of DerivedClass2, the return value is an instance of MyClass. Using @@species, any derived class can determine what type of value should be returned when a method returns an instance. And since Array uses @@species, you can make that change in a derived array class, such as:

```
class MyArray extends Array {
    static get [Symbol.species]() {
        return Array;
    }
}

let items = new MyArray(1, 2, 3, 4),
    subitems = items.slice(1, 3);

console.log(items instanceof MyArray);       // true
console.log(subitems instanceof Array);      // true
console.log(subitems instanceof MyArray);   // false
```

This code overrides @@species on MyArray, which inherits from Array. All of the inherited methods that return arrays will now use an instance of Array instead of MyArray.

In general, you should use the @@species property whenever you might want to use this.constructor in a class method. Doing so allows derived classes to override the return type easily. Additionally, if you are creating derived classes from a class that has @@species defined, be sure to use that value instead of the constructor.

new.target

In Chapter 3, you learned about new.target and how its value changes depending on how a function is called. You can also use new.target in class constructors to determine how the class is being invoked. In the simple case, new.target is equal to the constructor function for the class, as in this example:

```
class Rectangle {
    constructor(length, width) {
        console.log(new.target === Rectangle);
        this.length = length;
        this.width = width;
    }
}

// new.target is Rectangle
var obj = new Rectangle(3, 4);       // outputs true
```

In this code, you can see that new.target is equivalent to Rectangle when new Rectangle(3, 4) is called. Since class constructors cannot be called without new, new.target is always defined inside of class constructors. However, the value may not always be the same.

```
class Rectangle {
    constructor(length, width) {
        console.log(new.target === Rectangle);
        this.length = length;
        this.width = width;
    }
}

class Square extends Rectangle {
    constructor(length) {
        super(length, length)
    }
}

// new.target is Square
var obj = new Square(3);      // outputs false
```

Here, Square is calling the Rectangle constructor, so new.target is equal to Square when the Rectangle constructor is called. This is important because it gives each constructor the ability to alter its behavior based on how it's being called. For instance, you can create an abstract base class (one that cannot be instantiated directly) by using new.target:

```
// abstract base class
class Shape {
    constructor() {
        if (new.target === Shape) {
            throw new Error("This class cannot be instantiated directly.")
        }
    }
}

class Rectangle extends Shape {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }
}

var x = new Shape();                // throws error

var y = new Rectangle(3, 4);        // no error
console.log(y instanceof Shape);    // true
```

In this example, the `Shape` class constructor throws an error whenever `new.target` is `Shape`, meaning that `new Shape()` always throws an error. However, you can still use `Shape` as a base class, which is what `Rectangle` does. The `super()` call executes the `Shape` constructor and `new.target` is equal to `Rectangle` so the constructor continues without error.

Since classes cannot be called without `new`, `new.target` is never `undefined` inside of a class constructor.

Summary

For those who have struggled to understand JavaScript in the absence of classes, ECMAScript 6 classes provide an easier way to become acclimated with the language without needing to completely throw away their understanding of inheritance. ECMAScript 6 classes start out as syntactic sugar for the classical inheritance model of ECMAScript 5, but add a lot of features to reduce mistakes.

ECMAScript 6 classes work with prototypal inheritance by defining non-static methods on the class prototype while static methods end up on the constructor itself. All methods are non-enumerable, which better matches the behavior of built-in objects for which methods are typically nonenumerable by default. Additionally, class constructors cannot be called without `new`, ensuring that you can't accidentally call a class as a function.

Class-based inheritance allows you to derive a class from another class, function, or expression. This ability means you can call a function to determine the correct base to inherit from, allowing you to use mixins and other different composition patterns to create a new class. Inheritance works in such a way that inheriting from built-in objects, such as `Array`, is now possible and works as expected.

You can use `new.target` in class constructors to behave differently depending on how the class is called. The most common use is to create an abstract base class that throws an error when instantiated directly but still allows inheritance via other classes.

Overall, classes are an important addition to the language that provide a more concise syntax and better functionality for defining custom object types in a safe, consistent manner.

Arrays

Arrays have been one of the foundational JavaScript objects since the language's early days. Unfortunately, arrays remained the same for most of their existence until ECMAScript 5 introduced some new array methods. ECMAScript 6 continues the trend by updating arrays with a lot more functionality.

Creating Arrays

Prior to ECMAScript 6, the two primary ways of creating arrays were the `Array` constructor and array literal syntax. Both approaches require you to list out the array items individually and are otherwise fairly limited. If you had an array-like object (an object with numeric indices and a `length` property) and wanted to convert it into an array, your options were fairly limited and often required extra code. While these limitations might seem small, they turned out to be a growing pain point for large JavaScript applications that do a lot of array manipulation. To makes things easier, ECMAScript 6 adds two new methods: `Array.of()` and `Array.from()`.

Array.of()

JavaScript has long had a quirk around creating arrays with the `Array` constructor. The behavior of `new Array()` behaves differently based on the type and number of arguments passed into it. For example:

```
let items = new Array(1, 2);        // length is 2
console.log(items.length);          // 2
console.log(items[0]);              // 1
console.log(items[1]);              // 2

items = new Array(2);
console.log(items.length);          // 2
console.log(items[0]);              // undefined
console.log(items[1]);              // undefined

items = new Array(3, "2");
console.log(items.length);          // 2
console.log(items[0]);              // 3
console.log(items[1]);              // "2"

items = new Array("2");
console.log(items.length);          // 1
console.log(items[0]);              // "2"
```

When the `Array` constructor is passed a single numeric value, that value is set to be the `length` of the array; if a single non-numeric value is passed, then that value becomes the one and only item in the array; if multiple values are passed (numeric or not), then those values become items in the array. This behavior is both confusing and risky, as you may not always be aware of the type of data being passed.

ECMAScript 6 introduces `Array.of()` to solve this problem. `Array.of()` works in a manner that is similar to the `Array` constructor. The only difference is the removal of the special case regarding a single numeric value. The `Array.of()` method always creates an array containing its arguments regardless of the number of arguments or the argument types. Here are some examples:

```
let items = Array.of(1, 2);         // length is 2
console.log(items.length);          // 2
console.log(items[0]);              // 1
console.log(items[1]);              // 2

items = Array.of(2);
console.log(items.length);          // 1
console.log(items[0]);              // 2

items = Array.of("2");
console.log(items.length);          // 1
console.log(items[0]);              // "2"
```

The Array.of() method is similar to using an array literal, which is to say, you can use an array literal instead of Array.of() for native arrays most of the time. If you ever need to pass the Array constructor into a function, then you might want to pass Array.of() instead to get consistent behavior. For example:

```
function createArray(arrayCreator, value) {
    return arrayCreator(value);
}

let items = createArray(Array.of, value);
```

In this code, the createArray() function accepts an array creator function and a value to insert into the array. You can then pass Array.of as the first argument to createArray() to create a new array. It would be dangerous to pass Array directly if you cannot guarantee that value won't be a number.

The Array.of() method does not use the @@species property (discussed in Chapter 9) to determine the type of return value. Instead, it uses the current constructor (this inside of of()) to determine the correct data type to return.

Array.from()

One of the more cumbersome tasks in JavaScript has long been converting nonarray objects into actual arrays. For instance, if you have an arguments object (which is array-like) and want to work with it as if it's an array, then you'd need to convert it first. In ECMAScript 5, you'd write a function such as:

```
function makeArray(arrayLike) {
    var result = [];

    for (var i = 0, len = arrayLike.length; i < len; i++) {
        result.push(arrayLike[i]);
    }

    return result;
}

function doSomething() {
    var args = makeArray(arguments);

    // use args
}
```

This approach manually creates an array and copies over each item from `arguments` into that new array. While that works, it's a decent amount of code to perform a relatively simple operation. As such, developers soon discovered that you could shorten the amount of code by using the native array `slice()` method on array-like objects:

```
function makeArray(arrayLike) {
    return Array.prototype.slice.call(arrayLike);
}

function doSomething() {
    var args = makeArray(arguments);

    // use args
}
```

Even though this required less typing, it's not at all obvious that `Array.prototype.slice.call()` means "convert to an array." This works because you're setting the `this` value for `slice()` to the array-like object. Since `slice()` needs only numeric indices and a `length` property to function correctly, any array-like object will work.

The `Array.from()` method was added in ECMAScript 6 as a more obvious way of converting objects into arrays. You can pass either an iterable or an array-like object as the first argument and `Array.from()` returns an array. Here's a simple example:

```
function doSomething() {
    var args = Array.from(arguments);

    // use args
}
```

The Array.from() call in this example creates a new array based on the items in arguments. So args is an instance of Array that contains the same values in the same positions as arguments.

Array.from() also uses this to determine the type of array to return.

Mapping Conversion

If you want to take this conversion a step further, you can provide a second argument to Array.from() that is a mapping function used to convert each value into a final form. For example:

```
function translate() {
    return Array.from(arguments, (value) => value + 1);
}

let numbers = translate(1, 2, 3);

console.log(numbers);               // 2,3,4
```

Here, Array.from() is used with a mapping function that adds one to each item in the array. If the mapping function is on an object, you can optionally pass a third argument to Array.from() that represents the this value for the mapping function, such as:

```
let helper = {
    diff: 1,

    add(value) {
        return value + this.diff;
    }
};

function translate() {
    return Array.from(arguments, helper.add, helper);
}

let numbers = translate(1, 2, 3);

console.log(numbers);               // 2,3,4
```

This example uses the helper.add() method as the mapping function for the conversion. Since helper.add() uses this.diff, you need to provide the third argument to Array.from() specifying the value of this. In this way, Array.from() can easily handle conversion of data without needing to use bind() or another way of specifying the this value.

Use on Iterables

The Array.from() method works on both array-like objects and iterables. That means any object with an @@iterator property can be converted into an array using Array.from(). For example:

```
let numbers = {
    *[Symbol.iterator]() {
        yield 1;
        yield 2;
        yield 3;
    }
};

let numbers2 = Array.from(numbers, (value) => value + 1);

console.log(numbers2);              // 2,3,4
```

In this code, the numbers object is an iterable so it can be passed directly to Array.from() to convert its values into an array. The mapping function adds one to each number so the resulting array contains 2, 3, and 4 instead of 1, 2, and 3.

If an object is both array-like and iterable, then the iterator is used by `Array.from()` to determine the values to convert.

New Methods

Continuing the trend from ECMAScript 5, ECMAScript 6 adds several new methods to arrays. While the first two methods, `find()` and `findIndex()`, were meant to aid all developers, the others, `fill()` and `copyWithin()` are inspired largely by use cases for typed arrays (discussed later in this chapter).

The find() and findIndex() Methods

Prior to ECMAScript 5, searching through arrays was cumbersome because there were no builtin methods for doing so. ECMAScript 5 added `indexOf()` and `lastIndexOf()`, finally allowing developers to search for specific values inside of an array. As big of an improvement as these two methods were, they were still fairly limited because you could only search for one value at a time (meaning if you wanted to find the first even number in a series of numbers, for example, you'd need to write your own code to do so). ECMAScript 6 solved that problem by introducing two new methods: `find()` and `findIndex()`.

The `find()` and `findIndex()` methods work the same way. They both accepts two arguments, a callback function and an optional value to use for `this` inside of that function. The callback function is passed an array element, the index of that element in the array, and the array itself (same arguments as methods such as `map()` and `forEach()`) and should return `true` if the given value matches some criteria. Both `find()` and `findIndex()` stop searching the array the first time the callback function returns `true`, and the only difference is that `find()` returns the value whereas `findIndex()` returns the index at which the value was found. Here's an example:

```
let numbers = [25, 30, 35, 40, 45];

console.log(numbers.find(n => n > 33));        // 35
console.log(numbers.findIndex(n => n > 33));   // 2
```

This code uses both `find()` and `findIndex()` to locate the first value in the `numbers` array that is greater than 33. The call to `find()` returns 35 while `findIndex()` returns 2, the location of 35 in the `numbers` array.

Both find() and findIndex() are useful to find an array element that matches a condition rather than a value. If you only want to find a value, then indexOf() and lastIndexOf() are better choices.

The fill() Method

The fill() method fills one or more array elements with a specific value. When passed a value, fill() overwrites all of the values in an array with that value. For example:

```
let numbers = [1, 2, 3, 4];

numbers.fill(1);

console.log(numbers.toString());    // 1,1,1,1
```

Here, the call to numbers.fill(1) changes all of the values in numbers to 1. If you only want to change some of the elements, rather than all of them, you can optionally include a start index and an exclusive end index, such as:

```
let numbers = [1, 2, 3, 4];

numbers.fill(1, 2);

console.log(numbers.toString());    // 1,2,1,1

numbers.fill(0, 1, 3);

console.log(numbers.toString());    // 1,0,0,1
```

In this example, the last two array elements are filled with 1 by numbers.fill(1, 2) as 2 indicates the index at which to start filling elements. The end index is considered to be numbers.length because it isn’t specified. The next operation, numbers.fill(0, 1, 3), fills array elements at indices 1 and 2 with 0. In this way, you’re able to fill multiple array elements at once without overwriting the entire array.

If either the start or end index are negative, then those values are added to the array's length to determine the final location. For instance, a start location of -1 means that the index will be array.length-1 where array is the array on which fill() is called.

The copyWithin() Method

The copyWithin() method is similar to fill() in that it changes multiple array elements at the same time. However, instead of specifying a single value to assign to array elements, copyWithin() lets you copy array element values from the array itself. To accomplish that, you need to pass two arguments to copyWithin(), the index at which the values should be filled and the index starting at which values should be copied. For instance, if you want to copy the values from the first two elements in the array into the last two items in the array, you can do so as follows:

```
let numbers = [1, 2, 3, 4];

// paste values into array starting at index 2
// copy values from array starting at index 0
numbers.copyWithin(2, 0);

console.log(numbers.toString());    // 1,2,1,2
```

This code copies the values in numbers beginning from index 2, so both indices 2 and 3 will be overwritten. The second argument to copyWithin() is 0, which indicates to start copying values from index 0 and continue until there are no more elements to copy into.

By default, copyWithin() always copies values up to the end of the array, but you can provide an optional third argument to limit how many elements will be overwritten. That third argument is an exclusive end index at which copying of values stops. Here's an example:

```
let numbers = [1, 2, 3, 4];

// paste values into array starting at index 2
// copy values from array starting at index 0
// stop copying values when you hit index 1
numbers.copyWithin(2, 0, 1);

console.log(numbers.toString());    // 1,2,1,4
```

In this example, the optional end index is set to `1` so that only the value in index 0 is copied. The last element in the array remains unchanged.

As with `fill()`, if you pass a negative number for any argument to `copyWithin()`, the array's length is automatically added to that value to determine the index to use.

The use cases for `fill()` and `copyWithin()` may not be obvious to you at this point. That's because these methods originated on typed arrays and were then added to regular arrays for consistency. However, if you end up using typed arrays for manipulating the bits of a number, these methods become a lot more useful.

Typed Arrays

Typed arrays are special-purpose arrays designed to work with numeric types (not all types, as it may seem from the name). The origin of typed arrays can be traced back to WebGL, a port of Open GL ES 2.0 designed for use in web pages with the `<canvas>` element. Typed arrays were created as part of this port to provide fast bitwise arithmetic in JavaScript. The native JavaScript numbers were too slow to due to being stored in a 64-bit floating-point format and converted into 32-bit integers as needed, so typed arrays were introduced to circumvent this limitation and provide better performance for these operations. The concept is that any single number can be treated like arrays of bits, and in doing so, can make use of the familiar methods available on JavaScript arrays.

ECMAScript 6 adopted typed arrays as a formal part of the language to ensure better compatibility across JavaScript engines and interoperability with JavaScript arrays. While the ECMAScript 6 version of typed arrays is not exactly the same as the WebGL version, there are enough similarities to make the ECMAScript 6 version an evolution of the WebGL version rather than a different approach.

Numeric Data Types

JavaScript numbers are stored in IEEE 754 format, which uses 64 bits to store a floating-point representation of the number. This format represents both integers and floats in JavaScript, with conversion between the two formats happening frequently as numbers are changed. Typed arrays allow the storage and manipulation of eight different numeric types:

1. Signed 8-bit integer (int8)
2. Unsigned 8-bit integer (uint8)
3. Signed 16-bit integer (int16)
4. Unsigned 16-bit integer (uint16)
5. Signed 32-bit integer (int32)
6. Unsigned 32-bit integer (uint32)
7. 32-bit float (float32)
8. 64-bit float (float64)

If you want to represent an int8 today in a JavaScript number, you would be wasting 56 bits. Those bits might better be used to store additional int8's (or any other number that requires less than 56 bits). This is one of the use cases typed arrays address.

All of the operations and objects related to typed arrays are centered around these eight data types. In order to use them, though, you'll need to create an array buffer to store the data.

Array Buffers

The foundational piece underlying all typed arrays is an array buffer. An array buffer is a memory location for any number of bytes. Creating an array buffer is akin to calling something like `malloc()` in C to allocate memory without specifying what is contained within. You can create an array buffer by using the `ArrayBuffer` constructor and passing in the number of bytes it should contain:

```
let buffer = new ArrayBuffer(10);    // allocate 10 bytes
```

Once created, you can retrieve the number of bytes in the array buffer by using the `byteLength` property:

```
let buffer = new ArrayBuffer(10);    // allocate 10 bytes
console.log(buffer.byteLength);      // 10
```

The only other thing you can do is create a new array buffer that contains part of an existing array buffer using the slice() method. The slice() method works in a similar manner to the array slice() method in that you pass in the start index and end index as arguments and then return a new ArrayBuffer instance that is comprised of those elements from the original. For example:

```
let buffer = new ArrayBuffer(10);    // allocate 10 bytes


let buffer2 = buffer.slice(4, 6);
console.log(buffer2.byteLength);    // 2
```

In this code, buffer2 is created by extract the bytes at index 4 and 5 (the second argument to slice(), just like with arrays, is exclusive).

Of course, creating a storage location isn't very helpful without being able to write data into that location. To do so, you'll need to create a view.

Keep in mind, you cannot change the number of bytes that an array buffer represents. It always represents the exact number specified. You can change the data contained within an array buffer but never the size of the array buffer itself.

Views

While array buffers represent a memory location, views are the interface through which you manipulate that memory. A view operates on an array buffer, or a subset of an array buffer's bytes, reading and writing data in a particular format. The DataView type is a generic view on an array buffer that allows you to operate on all eight numeric data types. To do so, first create an ArrayBuffer and then use it to create a new DataView. Here's an example:

```
let buffer = new ArrayBuffer(10),
    view = new DataView(buffer);
```

The view object in this example has access to the entire 10 bytes of buffer. You can alternately create a view over just a portion of a buffer by providing a byte offset and, optionally, the number of bytes to include from that offset (defaults to the end of the buffer when not present). For example:

```
let buffer = new ArrayBuffer(10),
    view = new DataView(buffer, 5, 2);      // cover bytes 5 and 6
```

Here, `view` operates only on the bytes at indices 5 and 6. This approach allows you to create several views over the same array buffer, which can be useful if you want to have a single memory location for an entire application rather than dynamically allocating space as needed.

Retrieving View Information

You can retrieve information about the view by using the following read-only properties:

- `buffer` - the array buffer that the view is tied to
- `byteOffset` - the second argument to the `DataView` constructor if provided (0 by default)
- `byteLength` - the third argument to the `DataView` constructor if provided (the buffer's `byteLength` by default)

Using these properties, you can inspect exactly where a view is operating, such as:

```
let buffer = new ArrayBuffer(10),
    view1 = new DataView(buffer),           // cover all bytes
    view2 = new DataView(buffer, 5, 2);     // cover bytes 5 and 6

console.log(view1.buffer === buffer);       // true
console.log(view2.buffer === buffer);       // true
console.log(view1.byteOffset);              // 0
console.log(view2.byteOffset);              // 5
console.log(view1.byteLength);              // 10
console.log(view2.byteLength);              // 2
```

This code creates two views, `view1` that is a view over the entire array buffer and `view2` that operates on a small section of the array buffer. The `buffer` property for each view is the same because they both work on the same array buffer. The `byteOffset` and `byteLength` are different for each view, reflecting where in the array buffer the view operates.

Of course, reading information about memory isn't very useful on its own. You need to write data into and read data out of that memory to get any benefit.

Reading and Writing Data

There are two methods for each of the eight numeric data types, one to write data and one to read data. The methods have a name beginning with either "set" or "get" and followed by the data type abbreviation. The "set" methods accept three arguments, the byte offset at which to write, the value to write, and an optional boolean value indicating the value should be stored in little-endian format (with the least significant byte at byte 0 instead of in the last byte). The "get" methods accept two arguments, the byte offset to read from and an optional boolean value indicating the value should be read as little-endian. Here's an example:

```
let buffer = new ArrayBuffer(2),
    view = new DataView(buffer);

view.setInt8(0, 5);
view.setInt8(1, -1);

console.log(view.getInt8(0));       // 5
console.log(view.getInt8(1));       // -1
```

This example uses a two-byte array buffer to store two int8 values. The first value is set at offset 0 and the second at offset 1 reflecting that each is taking up a full byte (8 bits). Those values are later retrieved from their positions. While this example uses int8 values, you can use any of the eight numeric types with the corresponding methods. The complete list of methods is:

- `getInt8(byteOffset, littleEndian)` - read an int8 starting at `byteOffset`
- `setInt8(byteOffset, value, littleEndian)` - write an int8 starting at `byteOffset`
- `getUint8(byteOffset, littleEndian)` - read an uint8 starting at `byteOffset`
- `setUint8(byteOffset, value, littleEndian)` - write an uint8 starting at `byteOffset`
- `getInt16(byteOffset, littleEndian)` - read an int16 starting at `byteOffset`
- `setInt16(byteOffset, value, littleEndian)` - write an int16 starting at `byteOffset`
- `getUint16(byteOffset, littleEndian)` - read an uint16 starting at `byteOffset`
- `setUint16(byteOffset, value, littleEndian)` - write an uint16 starting at `byteOffset`
- `getInt32(byteOffset, littleEndian)` - read an int32 starting at `byteOffset`
- `setInt32(byteOffset, value, littleEndian)` - write an int32 starting at `byteOffset`
- `getUint32(byteOffset, littleEndian)` - read an uint32 starting at `byteOffset`
- `setUint32(byteOffset, value, littleEndian)` - write an uint32 starting at `byteOffset`

- getFloat32(byteOffset, littleEndian) - read a float32 starting at byteOffset
- setFloat32(byteOffset, value, littleEndian) - write a float32 starting at byteOffset
- getFloat64(byteOffset, littleEndian) - read a float64 starting at byteOffset
- setFloat64(byteOffset, value, littleEndian) - write a float64 starting at byteOffset

The interesting aspect of views is that you can read and write in any format at any point in time, regardless of how data was previously stored. For instance, what happens if you write two int8 values and read the buffer as int16? It works just fine, as in this example:

```
let buffer = new ArrayBuffer(2),
    view = new DataView(buffer);

view.setInt8(0, 5);
view.setInt8(1, -1);

console.log(view.getInt16(0));      // 1535
console.log(view.getInt8(0));       // 5
console.log(view.getInt8(1));       // -1
```

The call to view.getInt16(0) reads all of the bytes in the view and interprets those bytes as the number 1535. To understand why this happens, take a look at what each line does to the array buffer.

```
new ArrayBuffer(2)     0000000000000000
view.setInt8(0, 5);    0000010100000000
view.setInt8(1, -1);   0000010111111111
```

The array buffer starts out with 16 bits that are all zero. Adding the int8 value of 5 at the start of the array buffer introduces a couple of ones by writing the 8-bit representation (00000101). When -1 is written to the second byte, it sets all bits to one (the two's complement representation). At the end, the array buffer contains 16 bits that are then read out as a 16-bit integer using getInt16(). The interpretation of those 16 bits as a single number is 1535.

The DataView object is perfect for use cases that mix different data types in this way. However, if you're only using one specific data type, then the type-specific views are a better choice.

Type-Specific Views

ECMAScript 6 typed arrays are actually type-specific views for array buffers. Instead of using a generic DataView object to operate on an array buffer, you can use objects that enforce specific data types. There are nine type-specific views, corresponding to the eight numeric data types plus one additional optional for uint8 values. The following table is an abbreviated version of the one found in the specification (section 22.2) and lists out the various types:

Constructor Name	Element Size	Description	Equivalent C Type
Int8Array	1	8-bit 2's complement signed integer	signed char
Uint8Array	1	8-bit unsigned integer	unsigned char
Uint8ClampedArray	1	8-bit unsigned integer (clamped conversion)	unsigned char
Int16Array	2	16-bit 2's complement signed integer	short
Uint16Array	2	16-bit unsigned integer	unsigned short
Int32Array	4	32-bit 2's complement signed integer	int
Uint32Array	4	32-bit unsigned integer unsigned	int

Constructor Name	Element Size	Description	Equivalent C Type
Float32Array	4	32-bit IEEE floating point	float
Float64Array	8	64-bit IEEE floating point	double

The Uint8ClampedArray is the same as Uint8Array except when values are less than 0 or greater than 255. In that case, a Uint8ClampedArray will convert values lower than 0 to be 0 (-1 becomes 0, for example) and values higher than 255 to be 255 (300 becomes 255, for example).

Each of the typed arrays limits operations to working on a particular type of data, so all operations on Int8Array use int8 values. That means each typed array also has a difference byte size per element. Whereas Int8Array has a single byte element, Float64Array uses eight bytes per element. The elements are accessed using numeric indices just like regular arrays, allowing you to avoid the somewhat awkward calls to the “set” and “get” methods of DataView.

While typed arrays looks and behave similar to JavaScript arrays, they do not inherit from Array.

Creating Type-Specific Views

Typed array constructors accept multiple different types of arguments. First, you can create a new typed array by passing the same arguments as you would to DataView, meaning an array buffer, an optional byte offset, and an optional byte length. For example:

```
let buffer = new ArrayBuffer(10),
    view1 = new Int8Array(buffer),
    view2 = new Int8Array(buffer, 5, 2);

console.log(view1.buffer === buffer);       // true
console.log(view2.buffer === buffer);       // true
console.log(view1.byteOffset);              // 0
console.log(view2.byteOffset);              // 5
console.log(view1.byteLength);              // 10
console.log(view2.byteLength);              // 2
```

In this code, the two views are both UInt8Array instances that use buffer. Both view1 and view2 have the same buffer, byteOffset, and byteLength properties that exist on DataView instances. It's easy to swap in using a typed array whenever you use a DataView so long as you only work with one numeric type.

The second way to create a typed array is to pass a single number to the constructor. That number represents the number of elements (not bytes) to allocate to the array. In doing so, the constructor creates a new buffer that has the correct number of bytes to represent the number of array elements. You can then access the number of elements in the array by using the length property. For example:

```
let ints = new Int16Array(2),
    floats = new Float32Array(5);

console.log(ints.byteLength);       // 4
console.log(ints.length);           // 2

console.log(floats.byteLength);     // 20
console.log(floats.length);         // 5
```

The ints array is created to have two elements. Each 16-bit integer requires two bytes per value, so the array is allocated four bytes. The floats array is created to have five elements, so the number of bytes required is 20 (four bytes per element). In both cases, a new buffer is created and can be accessed using the buffer property if necessary.

⚠ If no argument is passed to a typed array constructor, the constructor acts as if 0 was passed. This effectively creates a typed array that cannot hold any data because zero bytes are allocated to the buffer.

Element Size

Each typed array is made up of a number of elements, and the element size is the number of bytes each element represents. This value is stored on a BYTES_PER_ELEMENT property on each constructor and each instance, so you can easily query the element size:

```
console.log(UInt8Array.BYTES_PER_ELEMENT);      // 1
console.log(UInt16Array.BYTES_PER_ELEMENT);     // 2

let ints = new Int8Array(5);
console.log(ints.BYTES_PER_ELEMENT);            // 1
```

The third way to create a typed array is to pass an object as the only argument to the constructor. The object can be any of the following:

- **Typed Array** - each element is copied into a new element on the new typed array (for example, an int8 is copied into an int16). The new typed array has a different array buffer than the one that was passed in.
- **Iterable** - the iterator is called to retrieve the items to insert into the typed array. The constructor will throw an error if any of the elements are invalid for the type.
- **Array** - the elements of the array are copied into a new typed array. The constructor will throw an error if any of the elements are invalid for the type.
- **Array-Like Object** - behaves the same as an array.

In each of these cases, a new typed array is created with the data from the source object. This can be especially useful when you want to initialize a typed array with some values, such as:

```
let ints1 = new Int16Array([25, 50]),
    ints2 = new Int32Array(ints1);

console.log(ints1.buffer === ints2.buffer);     // false

console.log(ints1.byteLength);      // 4
console.log(ints1.length);          // 2
console.log(ints1[0]);              // 25
console.log(ints1[1]);              // 50

console.log(ints2.byteLength);      // 8
console.log(ints2.length);          // 2
console.log(ints2[0]);              // 25
console.log(ints2[1]);              // 50
```

This example creates an `Int16Array` and initializes it with an array of two values. Then, an `Int32Array` is created and passed the `Int16Array`. The values 25 and 50 are copied from `ints1` into `ints2` as the two typed arrays have completely separate buffers. The same numbers are represented in both typed arrays but ints2 has eight bytes to represent the data while ints1 has only four.

Similarities with Arrays

As you've already seen, typed arrays can be used like regular arrays in many situations. You can see how many elements are in the array using the length property and you can access the elements directly using numeric indices, such as:

```
let ints = new Int16Array([25, 50]);

console.log(ints.length);          // 2
console.log(ints[0]);             // 25
console.log(ints[1]);             // 50

ints[0] = 1;
ints[1] = 2;

console.log(ints[0]);             // 1
console.log(ints[1]);             // 2
```

In this example, a new Int16Array with two items is created. The items are read from and written to using their numeric indices, and those values are automatically stored and converted into int16 values as part of the operation.

Typed arrays are also similar to regular arrays due to the availability of a large number of array methods. Here are the array methods you can use on typed arrays:

- copyWithin()
- entries()
- fill()
- filter()
- find()
- findIndex()
- forEach()
- indexOf()
- join()
- keys()
- lastIndexOf()

- map()
- reduce()
- reduceRight()
- reverse()
- slice()
- some()
- sort()
- values()

Keep in mind that while all of these methods act the same as those on Array.prototype, they are not the same methods. The typed array methods have additional checks for numeric type safety and, when an array is returned, will return a typed array instead of a regular array. Here's a simple example:

```
let ints = new Int16Array([25, 50]),
    mapped = ints.map(v => v * 2);

console.log(mapped.length);        // 2
console.log(mapped[0]);            // 50
console.log(mapped[1]);            // 100

console.log(mapped instanceof Int16Array);  // true
```

This example uses the map() method to create a new array based on the values in ints. The mapping function doubles each value in the array and returns a new Int16Array.

Also note that typed arrays have the same three iterators as regular arrays: entries(), keys(), and values(). These allow you to use the spread operator and for-of loops in the same way as you would regular arrays. For example:

```
let ints = new Int16Array([25, 50]),
    intsArray = [...ints];

console.log(intsArray instanceof Array);    // true
console.log(intsArray[0]);                  // 25
console.log(intsArray[1]);                  // 50
```

This code creates a new array intsArray containing the same data as the typed array ints. As with other iterables, the spread operator is an easy way to convert typed arrays into regular arrays.

Lastly, all typed arrays have static `of()` and `from()` methods that work the same way as `Array.of()` and `Array.from()`. The only difference is that the result is a typed array instead of a regular array. Otherwise, you can use these methods in the same way to create various typed arrays, such as:

```
let ints = Int16Array.of(25, 50),
    floats = Float32Array.from([1.5, 2.5]);

console.log(ints instanceof Int16Array);        // true
console.log(floats instanceof Float32Array);    // true

console.log(ints.length);       // 2
console.log(ints[0]);           // 25
console.log(ints[1]);           // 50

console.log(floats.length);     // 2
console.log(floats[0]);         // 1.5
console.log(floats[1]);         // 2.5
```

The `of()` and `from()` methods in this example are used to create an `Int16Array` and `Float32Array`, respectively. These methods ensure that typed arrays can be created just as easily as regular arrays.

Differences from Arrays

The most importance difference between typed arrays and regular arrays is that typed arrays are not regular arrays. That means they do not inherit from `Array` and `Array.isArray()` returns `false` when passed a typed array. For example:

```
let ints = new Int16Array([25, 50]);

console.log(ints instanceof Array);     // false
console.log(Array.isArray(ints));       // false
```

The `ints` variable is a typed array, so it's not an instance of `Array` and cannot otherwise be identifier as an array. This distinction is important because there are many ways in which typed arrays do not act like regular arrays.

Whereas regular arrays can grow and shrink as you interact with them, typed arrays always remain the same size. You cannot assign a value to a nonexistent numeric index like you can with regular arrays, as typed arrays will ignore the operation. Here's an example:

```
let ints = new Int16Array([25, 50]);

console.log(ints.length);          // 2
console.log(ints[0]);              // 25
console.log(ints[1]);              // 50

ints[2] = 5;

console.log(ints.length);          // 2
console.log(ints[2]);              // undefined
```

Despite assigning to the numeric index 2 in this example, the ints array does not grow at all. The length remains the same and the value is thrown away.

Typed arrays also have checks to ensure that only valid data types are used. Zero is used in place of any invalid values. For example:

```
let ints = new Int16Array(["hi"]);

console.log(ints.length);       // 1
console.log(ints[0]);           // 0
```

This code attempts to use the string value "hi" in an Int16Array. Of course, strings are invalid data types in typed arrays, so the value is inserted as zero instead. The length of the array is still one, and the ints[0] slot exists, it's just filled with zero instead of the string. The same restriction applies to all methods that modify values in a typed array.For example, if the function passed to map() returns an invalid value for the type array, then zero is used instead:

```
let ints = new Int16Array([25, 50]),
    mapped = ints.map(v => "hi");

console.log(mapped.length);        // 2
console.log(mapped[0]);            // 0
console.log(mapped[1]);            // 0

console.log(mapped instanceof Int16Array);  // true
console.log(mapped instanceof Array);       // false
```

Since the string value "hi" isn't a 16-bit integer, it's replaced with 0 in the resulting array. All of the array methods have similar error correction behavior to avoid throwing errors when invalid data is present.

The last difference between typed arrays and regular arrays is that typed arrays are missing several array methods. The following methods are not available on typed arrays:

- concat()
- pop()
- push()
- shift()
- splice()
- unshift()

With the exception of concat(), the other methods can change the size of an array and so are not available for typed arrays (since they cannot change size). The concat() method isn't available because it is unclear what concatenating two typed arrays, especially if they dealt with different data types, would mean for the result.

Additional Methods

There are a couple of typed arrays methods that are not present on regular arrays: set() and subarray(). These two methods are opposites in that set() allows you to copy another array into an existing typed array whereas subarray() let's you extract part of an existing typed array into a new typed array.

The set() method accepts an array (either typed or regular) and an optional offset at which to insert the data (default to zero). The data from the array argument is copied into the destination typed array while ensuring only valid data types are used. Here's an example:

```
let ints = new Int16Array(4);

ints.set([25, 50]);
ints.set([75, 100], 2);

console.log(ints.toString());    // 25,50,75,100
```

This code creates an Int16Array with four elements. The first call to set() copies two values to the first and second elements in the array. The second call to set() uses an offset of 2 to indicate that the values should be placed in the array starting at the third element.

Whereas set() inserts new values into a typed array, subarray() extracts values into a new typed array. The subarray() method accepts an optional start and end index (the end index is exclusive, as in methods like slice()) and returns a new typed array. You can also omit both arguments to create a clone of the typed array. For example:

```
let ints = new Int16Array([25, 50, 75, 100]),
    subints1 = ints.subarray(),
    subints2 = ints.subarray(2),
    subints3 = ints.subarray(1, 3);

console.log(subints1.toString());   // 25,50,75,100
console.log(subints2.toString());   // 75,100
console.log(subints3.toString());   // 50,75
```

There are three typed arrays created from the original ints array in this example. The subints1 array is a clone of ints, containing all the same information. The subints2 array starts copying data from index 2, and so contains only the last two elements of the array (75 and 100). The subints3 array contains only the middle two elements of the ints array, as both arguments were used for subarray().

Summary

ECMAScript 6 continues the work of ECMAScript 5 by continuing to update and change arrays to be more useful. There are now two new ways to create arrays, Array.of() and Array.from(). These methods can each be used to create arrays and, in the case of Array.from(), convert iterables and arraylike objects into arrays. Both methods are inherited by derived array classes and use the @@species property to determine what type of value should be returned.

There are also several new methods on arrays. The fill() and copyWithin() methods allow you to alter array elements in-place. The find() and findIndex() methods are useful for finding the first element in an array that matches some criteria. The former returns the first element that fits the criteria and the latter returns the index at which the element is found.

Typed arrays are not actually arrays, as they do not inherit from Array, but do look and behave a lot like arrays. Typed arrays contain one of eight different numeric data types and are built upon ArrayBuffer objects that represent the underlying bits of a number or series of numbers. Typed arrays are a more efficient way of doing bitwise arithmetic because the values are not converted back and forth between formats, as is the case with the JavaScript number type.

Promises

One of the most powerful aspects of JavaScript is how easily it handles asynchronous programming. Since JavaScript originated as a language for the web, it was a requirement to be able to respond to asynchronous user interactions such as clicks and key presses. Node.js further popularized asynchronous programming in JavaScript by using callbacks as an alternative to events. As more and more programs started using asynchronous programming, there was a growing sense that these two models, events and callbacks, weren't powerful enough to support everything that developers wanted to do. Promises are the solution to this problem.

Promises are another option for asynchronous programming, and similar functionality is available in other languages under names such as futures and deferreds. The basic idea is to specify some code to be executed later (as with events and callbacks) and also explicitly indicate if the code succeeded or failed in its job. In that way, you can chain promises together based on success or failure in ways that are easier to understand and debug.

Before you can get a good understanding of how promises work, however, it's important to understand some of the basic concepts upon which they are built.

Asynchronous Programming Background

JavaScript engines are built on the concept of a single-threaded event loop. Single-threaded means that only one piece of code is executed at any given point in time. This stands in contrast to other languages such as Java or C++ that may use threads to allow multiple different pieces of code to execute at the same time. Maintaining and protecting state when multiple pieces of code can access and change that state is a difficult problem and the source of frequent bugs in thread-based software.

Because JavaScript engines can only execute one piece of code at a time, it's necessary to keep track of code that is meant to run. That code is kept in a job queue. Whenever a piece of code is ready to be executed, it is added to the job queue. When the JavaScript engine is finished executing code, the event loop picks the next job in the queue and executes it. The event loop is a process inside of the JavaScript engine that monitors code execution and manages the job queue. Keep in mind that as a queue, job execution runs from the first job in the queue to the last.

Events

When a user clicks a button or presses a key on the keyboard, an event is triggered (such as `onclick`). That event may be used to respond to the interaction by adding a new job to the back of the job queue. This is the most basic form of asynchronous programming JavaScript has: the event handler code doesn't execute until the event fires, and when it does execute, it has the appropriate context. For example:

```
let button = document.getElementById("my-btn");
button.onclick = function(event) {
    console.log("Clicked");
};
```

In this code, `console.log("Clicked")` will not be executed until `button` is clicked. When `button` is clicked, the function assigned to `onclick` is added to the back of the job queue and will be executed when all other jobs ahead of it are complete.

Events work well for simple interactions such as this, but chaining multiple separate asynchronous calls together becomes more complicated because you must keep track of the event target (`button` in the previous example) for each event. Additionally, you need to ensure all appropriate event handlers are added before the first time an event occurs. For instance, if `button` in the previous example was clicked before `onclick` is assigned, then nothing would happen.

So while events are useful for responding to user interactions and similar functionality that occurs infrequently, they aren't very flexible for more complex needs.

Callbacks

When Node.js was created, it furthered the asynchronous programming model by popularizing the callback pattern. The callback pattern is similar to the event model because it doesn't execute code until a later point in time; it is different because the function to call is passed in as an argument. For example:

```
readFile("example.txt", function(err, contents) {
    if (err) {
        throw err;
    }

    console.log(contents);
});
console.log("Hi!");
```

This example uses the traditional Node.js style of error-first callback. The `readFile()` function is intended to read from a file on disk (specified as the first argument) and then execute the callback (the second argument) when complete. If there’s an error, the err argument of the callback is an error object; otherwise, the contents argument contains the file contents as a string.

Using the callback pattern, readFile() begins executing immediately and pauses when it begins reading from the disk. That means console.log("Hi!") is output immediately after readFile() is called (before console.log(contents)). When readFile() has finished, it adds a new job to the end of the job queue with the callback function and its arguments. That job is then executed upon completion of all other jobs ahead of it.

The callback pattern is more flexible than events because it is easier to chain multiple calls together. For example:

```
readFile("example.txt", function(err, contents) {
    if (err) {
        throw err;
    }

    writeFile("example.txt", function(err) {
        if (err) {
            throw err;
        }

        console.log("File was written!");
    });
});
```

In this code, a successful call to readFile() results in another asynchronous call, this time to writeFile(). Note that the same basic pattern of checking err is present in both functions. When readFile() is complete, it adds a job to the job queue that results in writeFile() being called (assuming no errors). Then, writeFile() adds a job to the job queue when it is complete.

While this works fairly well, you can quickly get into a pattern that has come to be known as callback hell. Callback hell occurs when you nest too many callbacks:

```
method1(function(err, result) {

    if (err) {
        throw err;
    }

    method2(function(err, result) {

        if (err) {
            throw err;
        }

        method3(function(err, result) {

            if (err) {
                throw err;
            }

            method4(function(err, result) {

                if (err) {
                    throw err;
                }

                method5(result);
            });

        });

    });

});
```

Nesting multiple method calls, as in this example, creates a tangled web of code that is hard to understand and debug.

Callbacks also present problems when you want to accomplish more complex functionality. What if you'd like two asynchronous operations to run in parallel and be notified when they both are complete? What if you'd like to kick off two asynchronous operations but only take the first one to complete? In these cases, you end up needing to keep track of multiple callbacks and cleanup operations. This is precisely where promises greatly improve the situation.

Promise Basics

A promise is a placeholder for the result of an asynchronous operation. Instead of subscribing to an event or passing a callback to a function, the function can return a promise, such as:

```
// readFile promises to complete at some point in the future
let promise = readFile("example.txt");
```

In this code, `readFile()` doesn't actually start reading the file immediately (that will happen later). It returns a promise object that represents the asynchronous operation so you can work with it later. Before using a promise, though, it's important to understand a promise lifecycle and how that lifecycle affects your results.

Lifecycle

Each promise goes through a short lifecycle starting in the pending state, which is an indicator that the asynchronous operation has not yet completed. The promise in the last example is in the pending state as soon as it is returned from `readFile()`. Once the asynchronous operation completes, the promise is considered settled and enters one of two possible states:

1. Fulfilled - the promise's asynchronous operation has completed successfully
2. Rejected - the promise's asynchronous operation did not complete successfully (either due to an error or some other cause)

You can't determine which state the promise is in programmatically, but you can take a specific action when a promise changes state by using the `then()` method.

There is an internal `[[PromiseState]]` property that is set to `"pending"`, `"fulfilled"`, or `"rejected"` to reflect the promise's state. This property is not exposed on promise objects.

The `then()` method is present on all promises and takes two arguments. The first argument is a function to call when the promise is fulfilled. Any additional data related to the asynchronous operation is passed into this fulfillment function. The second argument is a function to call when the promise is rejected. Similar to the fulfillment function, the rejection function is passed any additional data related to the rejection.

Any object that implements the then() method in this way is called a thenable, so all promises are thenables but all thenables are not promises.

Both arguments to then() are optional, so you can listen for any combination of fulfillment and rejection. For example:

```
let promise = readFile("example.txt");

// listen for both fulfillment and rejection
promise.then(function(contents) {
    // fulfillment
    console.log(contents);
}, function(err) {
    // rejection
    console.error(err.message);
});

// listen for just fulfillment - errors are not reported
promise.then(function(contents) {
    // fulfillment
    console.log(contents);
});

// listen for just rejection - success is not reported
promise.then(null, function(err) {
    // rejection
    console.error(err.message);
});
```

There is also a catch() method that behaves the same as then() when only a rejection handler is passed. For example:

```
promise.catch(function(err) {
    // rejection
    console.error(err.message);
});

// is the same as:

promise.then(null, function(err) {
    // rejection
    console.error(err.message);
});
```

The intent is to use a combination of `then()` and `catch()` to properly handle the result of asynchronous operations. The benefit of this over both events and callbacks is that it's completely clear whether the operation succeeded or failed. (Events tend not to fire when there's an error and in callbacks you must always remember to check the error argument.)

⚠ If you don't attach a rejection handler to a promise, all failures happen silently. It's a good idea to always attach a rejection handler even if it just logs the failure.

One of the unique aspects of promises is that a fulfillment or rejection handler will still be executed if it is added after the promise is already settled. This allows you to add new fulfillment and rejection handlers at any point in time and be assured that they will be called. For example:

```
let promise = readFile("example.txt");

// original fulfillment handler
promise.then(function(contents) {
    console.log(contents);

    // now add another
    promise.then(function(contents) {
        console.log(contents);
    });
});
```

In this example, the fulfillment handler adds another fulfillment handler to the same promise. The promise is already fulfilled at this point, so the new fulfillment handler is added to the job queue and called when ready. Rejection handlers work the same way in that they can be added at any point and are guaranteed to be called.

Each call to `then()` or `catch()` creates a new job to be executed when the promise is resolved. However, these jobs end up in a separate job queue that is reserved strictly for promises. The precise details of this second job queue aren’t all that important for understanding how to use promises so long as you understand how job queues work in general.

Creating Unsettled Promises

New promises are created using the `Promise` constructor. This constructor accepts a single argument, which is a function (called the executor) containing the code to initialize the promise. The executor is passed two functions as arguments, `resolve()` and `reject()`. The `resolve()` function is called when the executor has finished successfully in order to signal that the promise is ready to be resolved while the `reject()` function indicates that the executor has failed. Here’s an example using a promise in Node.js to implement the `readFile()` function from earlier in this chapter:

```
// Node.js example

let fs = require("fs");

function readFile(filename) {
    return new Promise(function(resolve, reject) {

        // trigger the asynchronous operation
        fs.readFile(filename, { encoding: "utf8" }, function(err, contents) {

            // check for errors
            if (err) {
                reject(err);
            }

            // the read succeeded
            resolve(contents);

        });
    });
}

let promise = readFile("example.txt");

// listen for both fulfillment and rejection
promise.then(function(contents) {
    // fulfillment
    console.log(contents);
}, function(err) {
    // rejection
    console.error(err.message);
});
```

In this example, the native Node.js `fs.readFile()` asynchronous call is wrapped in a promise. The executor either passes the error object to `reject()` or the file contents to `resolve()`.

Keep in mind that the executor runs immediately when `readFile()` is called. When either `resolve()` or `reject()` is called inside the executor, a job is added to the job queue in order to resolve the promise. This is called job scheduling, and if you've ever used `setTimeout()` or `setInterval()`, then you're already familiar with it. The idea is that a new job is added to the job queue so as to say, “don't execute this right now, but execute it later.” In the case of setTimeout() and setInterval(), you’re specifying a delay before the job is added to the queue:

```
// add this function to the job queue after 500ms have passed
setTimeout(function() {
    console.log("Timeout");
}, 500)

console.log("Hi!");
```

In this example, the code schedules a job to be added to the job queue after 500ms. That results in the following output:

```
Hi!
Timeout
```

You can tell from the output that the function passed to setTimeout() was executed after console.log("Hi!"). Promises work in a similar way.

The promise executor is executed immediately, meaning that it will execute prior to anything that appears after it in the source code. For example:

```
let promise = new Promise(function(resolve, reject) {
    console.log("Promise");
    resolve();
});

console.log("Hi!");
```

The output for this example is:

```
Promise
Hi!
```

When resolve() is called, that triggers an asynchronous operation. Functions passed to then() and catch() are executed asynchronously, as these will also be added to the job queue. Here’s an example:

```
let promise = new Promise(function(resolve, reject) {
    console.log("Promise");
    resolve();
});

promise.then(function() {
    console.log("Resolved.");
});

console.log("Hi!");
```

The output for this example is:

```
Promise
Hi!
Resolved
```

Note that even though the call to `then()` appears before `console.log("Hi!")`, it doesn't actually execute until later (unlike the executor). That's because fulfillment and rejection handlers are always added to the end of the job queue after the executor has completed.

Creating Settled Promises

The `Promise` constructor is the best way to create unsettled promises due to the dynamic nature of what the promise executor does. However, if you want a promise to represent just a single known value, then it doesn't make sense to go through the work of scheduling a job that simply passes a value to `resolve()`. Instead, there are two methods that create settled promises given a specific value.

The `Promise.resolve()` method accepts a single argument and returns a promise in the fulfilled state. That means there is no job scheduling that occurs and you need to add one or more fulfillment handlers to the promise to retrieve the value. For example:

```
let promise = Promise.resolve(42);

promise.then(function(value) {
    console.log(value);         // 42
});
```

This code creates a fulfilled promise so the fulfillment handler receives 42 as `value`. If a rejection handler were added to this promise, it would never be called because the promise will never be in the rejected state.

You can also create rejected promises by using the `Promise.reject()` method. This works in the same way as `Promise.resolve()` except that the created promise is in the rejected state. That means any additional rejection handlers added to the promise will be called but not the fulfillment handlers:

```
let promise = Promise.reject(42);

promise.catch(function(value) {
    console.log(value);         // 42
});
```

If you pass a promise to either `Promise.resolve()` or `Promise.reject()`, the promise is returned without modification.

Both `Promise.resolve()` and `Promise.reject()` also accept non-promise thenables as arguments and will create a new promise that is called after `then()`. A non-promise thenable is created when an object has a `then()` method that accepts two arguments: `resolve` and `reject`. For example:

```
let thenable = {
    then: function(resolve, reject) {
        resolve(42);
    }
};
```

The `thenable` object in this example has no characteristics associated with a promise other than the `then()` method. It can be converted into a fulfilled promise using `Promise.resolve()`:

```
let thenable = {
    then: function(resolve, reject) {
        resolve(42);
    }
};

let p1 = Promise.resolve(thenable);
p1.then(function(value) {
    console.log(value);     // 42
});
```

In this example, Promise.resolve() calls thenable.then() so that a promise state can be determined. Since this code calls resolve(42), the promise state for thenable is fulfilled. A new promise is created in the fulfilled state with the value passed from the thenable (42) so the fulfillment handler for p1 receives 42 as the value. The same process can be used with Promise.reject() in order to create a rejected promise from a thenable:

```
let thenable = {
    then: function(resolve, reject) {
        reject(42);
    }
};

let p1 = Promise.reject(thenable);
p1.catch(function(value) {
    console.log(value);     // 42
});
```

This example is similar to the last except that Promise.reject() is used on thenable. Doing so executes thenable.then() and creates a new promise in the rejected state with a value of 42. That value is then passed to the rejection handler for p1.

Both Promise.resolve() and Promise.reject() work in this way to allow you to easily work with non-promise thenables. Whenever you're unsure if an object is a promise, passing the object through Promise.resolve() or Promise.reject() (depending on your anticipated result) is the best approach since promises are just passed through without any change.

Executor Errors

If an error is thrown inside of an executor, then the promise's rejection handler is called. For example:

```
let promise = new Promise(function(resolve, reject) {
    throw new Error("Explosion!");
});

promise.catch(function(error) {
    console.log(error.message);     // "Explosion!"
});
```

In this code, the executor intentionally throws an error. There is an implicit try-catch inside of every executor such that the error is caught and then passed to the rejection handler. In effect, the previous example is equivalent to:

```
// equivalent of previous example
let promise = new Promise(function(resolve, reject) {
    try {
        throw new Error("Explosion!");
    } catch (ex) {
        reject(ex);
    }
});

promise.catch(function(error) {
    console.log(error.message);     // "Explosion!"
});
```

The executor handles catching any thrown errors in order to simplify this common use case. However, it also has the caveat that an error thrown in the executor is only reported when a rejection handler is present; otherwise, the error is suppressed. This became a problem for developers early on in the use of promises so JavaScript environments decided to address it by providing hooks for catching rejected promises.

Global Promise Rejection Handling

One of the most controversial aspects of promises is the silent failure that occurs when a promise is rejected and doesn't have a rejection handler. Some consider this the biggest flaw in the specification as it's the only part of the JavaScript language that doesn't make errors apparent when they occur.

Determining whether a promise rejection was handled isn't straightforward due to the nature of promises. A promise may be rejected and handled only at a later point in time, for example:

```
let rejected = Promise.reject(42);

// at this point, rejected is unhandled

// some time later...
rejected.catch(function(value) {
    // now rejected has been handled
    console.log(value);
});
```

Because you can call `then()` or `catch()` at any point and have them work correctly regardless of whether the promises is settled or not, it's hard to know precisely when a promise is going to be handled.

While it's possible that the next version of ECMAScript will address this problem, both browsers and Node.js have implemented changes to address this developer pain point. Note these are not part of the ECMAScript 6 specification but are valuable tools when using promises.

Node.js Rejection Handling

In Node.js, there are two events on the `process` object related to promise rejection handling:

- `unhandledRejection` - emitted when a promise is rejected and there is no rejection handler called within one turn of the event loop
- `rejectionHandled` - emitted when a promise is rejected and there is a rejection handler called after one turn of the event loop

These two events are designed to work together to help identify promises that are rejected and not handled.

The `unhandledRejection` event handler is passed two arguments: the rejection reason (frequently an error object) and the promise that was rejected. Here's a simple example:

```
let rejected;

process.on("unhandledRejection", function(reason, promise) {
    console.log(reason.message);            // "Explosion!"
    console.log(rejected === promise);      // true
});

rejected = Promise.reject(new Error("Explosion!"));
```

This example creates a rejected promise with an error object and listens for the `unhandledRejection` event. The event handler receives the error object as the first argument and the promise as the second.

The `rejectionHandled` event handler has only one argument, which is the promise that was rejected. For example:

```
let rejected;

process.on("rejectionHandled", function(promise) {
    console.log(rejected === promise);              // true
});

rejected = Promise.reject(new Error("Explosion!"));

// wait to add the rejection handler
setTimeout(function() {
    rejected.catch(function(value) {
        console.log(value.message);     // "Explosion!"
    });
}, 1000);
```

Here, the `rejectionHandled` event is emitted when the rejection handler is finally called. If the rejection handler was attached directly to `rejected` after its creation, then the event wouldn't have been emitted because the rejection handler would have been called during the same turn of the event loop in which `rejected` was created.

In order to properly track potentially unhandled rejections, you need to use both events to keep a list of potentially unhandled rejections and then wait some period of time to inspect the list. For example:

```
let possiblyUnhandledRejections = new Map();

// when a rejection is unhandled, add it to the map
process.on("unhandledRejection", function(reason, promise) {
    possiblyUnhandledRejections.set(promise, reason);
});

process.on("rejectionHandled", function(promise) {
    possiblyUnhandledRejections.delete(promise);
});

setInterval(function() {

    possiblyUnhandledRejections.forEach(function(reason, promise) {
        console.log(reason.message ? reason.message : reason);

        // do something to handle these rejections
        handleRejection(promise, reason);
    });

    possiblyUnhandledRejections.clear();

}, 60000);
```

This code is a simple unhandled rejection tracker. It uses a map to store promises and their rejection reasons where the promise is the key and the reason is the value. Each time unhandledRejection is emitted, the promise and its rejection reason is added to the map; each time rejectionHandled is emitted, the promise is removed from the map. As a result, possiblyUnhandledRejections continues to grow and shrink as each event is called. The setInterval() call periodically checks each the list of possible unhandled rejections and outputs the information to the console (in reality, you probably want to do something else to either log or otherwise handle the rejection). A map is used in this example instead of a weak map because you need to inspect the map periodically to see which promises are present, and that's not possible with a weak map.

While this example is specific to Node.js, browsers have implemented a similar mechanism for notifying developers about unhandled rejections.

Browser Rejection Handling

Browsers also emit two events to help identify unhandled rejections. These events are emitted by the window object and are effectively the same as their

Node.js equivalents:

- unhandledrejection - emitted when a promise is rejected and there is no rejection handler called within one turn of the event loop
- rejectionhandled - emitted when a promise is rejected and there is a rejection handler called after one turn of the event loop

The event handler for these events receives an event object with the following properties:

- type - the name of the event ("unhandledrejection" or "rejectionhandled")
- promise - the promise object that was rejected
- reason - the rejection value from the promise

Aside from using an event object instead of individual parameters in the event handler, the other difference in the browser implementation is that the rejection value (reason) is available for both events. For example:

```
let rejected;

window.onunhandledrejection = function(event) {
    console.log(event.type);                   // "unhandledrejection"
    console.log(event.reason.message);         // "Explosion!"
    console.log(rejected === event.promise);   // true
});

window.onrejectionhandled = function(event) {
    console.log(event.type);                   // "rejectionhandled"
    console.log(event.reason.message);         // "Explosion!"
    console.log(rejected === event.promise);   // true
});

rejected = Promise.reject(new Error("Explosion!"));
```

This code assigns both event handlers use the DOM Level 0 notation of onunhandledrejection and onrejectionhandled (you can also use addEventListener("unhandledrejection") and addEventListener("rejectionhandled") if you prefer). Each event handler receives an event object containing information about the rejected promise. All three properties, type, promise, and reason, are available in both event handlers.

The code to keep track of unhandled rejections in the browser is very similar to the code for Node.js:

```
let possiblyUnhandledRejections = new Map();

// when a rejection is unhandled, add it to the map
window.onunhandledrejection = function(event) {
    possiblyUnhandledRejections.set(event.promise, event.reason);
};

window.onrejectionhandled = function(event) {
    possiblyUnhandledRejections.delete(event.promise);
};

setInterval(function() {

    possiblyUnhandledRejections.forEach(function(reason, promise) {
        console.log(reason.message ? reason.message : reason);

        // do something to handle these rejections
        handleRejection(promise, reason);
    });

    possiblyUnhandledRejections.clear();

}, 60000);
```

This implementation is almost exactly the same as the Node.js implementation, the only real difference is where the information is retrieved from in the event handlers. Otherwise, this uses the same approach of storing promises and their rejection values in a map and then inspecting them later.

Handling promise rejections can be tricky, but it’s far from the only tricky concept involving promises. You’ve just begun to see how powerful promises can really be, and it’s time to take the next step and chain several promises together.

Chaining Promises

To this point, promises may seem like little more than an incremental improvement over using some combination of a callback and setTimeout(), but there is much more to promises than meets the eye. More specifically, there are a number of ways to chain promises together to accomplish more complex asynchronous behavior.

Each call to then() or catch() actually creates and returns another promise. This second promise is resolved only once the first has been fulfilled or rejected. For example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    console.log(value);
}).then(function() {
    console.log("Finished");
});
```

The output from this example is:

```
42
Finished
```

The call to p1.then() returns a second promise on which then() is called. The second then() fulfillment handler is only called after the first promise has been resolved. If you unchain this example, it looks like this:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

// same as

let p2 = p1.then(function(value) {
    console.log(value);
})

p2.then(function() {
    console.log("Finished");
});
```

As you might have guessed, p2.then() also returns a promise, but it's not used in this example.

Catching Errors

Promise chaining allows you to catch errors that may occur in a fulfillment or rejection handler from a previous promise. For example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    throw new Error("Boom!");
}).catch(function(error) {
    console.log(error.message);     // "Boom!"
});
```

In this example, the fulfillment handler for `p1` throws an error. The chained call to `catch()`, which is on a second promise, is able to receive that error through its rejection handler. The same is true if a rejection handler throws an error:

```
let p1 = new Promise(function(resolve, reject) {
    throw new Error("Explosion!");
});

p1.catch(function(error) {
    console.log(error.message);     // "Explosion!"
    throw new Error("Boom!");
}).catch(function(error) {
    console.log(error.message);     // "Boom!"
});
```

Here, the executor throws an error then triggers `p1`'s rejection handler. That handler then throws another error that is caught by the second promise's rejection handler. In this way, chained promise calls can be made aware of errors in other promises in the chain.

It's recommended to always have a rejection handler at the end of a promise chain to ensure that you can properly handle any errors that may occur.

Returning Values in Promise Chains

Another important aspect of promise chains is the ability to pass data from one promise to the next. You've already seen that a value passed to the `resolve()` handler inside an executor is passed to the fulfillment handler for that promise.

You can continue passing data along by specifying a return value from the fulfillment handler. For example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    console.log(value);         // "42"
    return value + 1;
}).then(function(value) {
    console.log(value);         // "43"
});
```

In this example, the fulfillment handler for `p1` returns a value (`value + 1`). Since `value` is 42 (from the executor) then the fulfillment handler returns 43. That value is then passed to the fulfillment handler of the second promise that can output it to the console.

The same thing is possible using the rejection handler. When a rejection handler is called, it has the option of returning a value. That value is then used to fulfill the next promise in the chain. For example:

```
let p1 = new Promise(function(resolve, reject) {
    reject(42);
});

p1.catch(function(value) {
    // first fulfillment handler
    console.log(value);         // "42"
    return value + 1;
}).then(function(value) {
    // second fulfillment handler
    console.log(value);         // "43"
});
```

Here, the executor calls `reject()` with 42. That value is passed into the rejection handler for the promise, where `value + 1` is returned. Even though this return value is coming from a rejection handler, it is still used in the fulfillment handler of the next promise in the chain. This allows for the failure of one promise to allow recovery of the entire chain if necessary.

Returning Promises in Promise Chains

Returning primitive values from fulfillment and rejection handlers allows passing of data between promises, but what if you return an object? If the object is a promise, then there’s an extra step that’s taken to determine how to proceed. Consider the following example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

p1.then(function(value) {
    // first fulfillment handler
    console.log(value);     // 42
    return p2;
}).then(function(value) {
    // second fulfillment handler
    console.log(value);     // 43
});
```

In this code, p1 schedules a job that resolves to 42. The fulfillment handler for p1 returns p2, a promise already in the resolved state. The second fulfillment handler is called because p2 has been fulfilled. If p2 was rejected, the second fulfillment handler would not be called and instead a rejection handler (if present) would be called.

The important thing to recognize about this pattern is that the second fulfillment handler is not added to p2, but rather to a third promise. It’s this third promise that the second fulfillment handler is attached to. The previous example is equivalent to this:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = p1.then(function(value) {
    // first fulfillment handler
    console.log(value);     // 42
    return p2;
});

p3.then(function(value) {
    // second fulfillment handler
    console.log(value);     // 43
});
```

Here, it’s clear that the second fulfillment handler is attached to `p3` rather than `p2`. This is a subtle but important distinction as the second fulfillment handler will not be called if `p2` is rejected. For example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

p1.then(function(value) {
    // first fulfillment handler
    console.log(value);     // 42
    return p2;
}).then(function(value) {
    // second fulfillment handler
    console.log(value);     // never called
});
```

In this example, the second fulfillment handler is never called because `p2` is rejected. You could, however, attach a rejection handler instead:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

p1.then(function(value) {
    // first fulfillment handler
    console.log(value);     // 42
    return p2;
}).catch(function(value) {
    // rejection handler
    console.log(value);     // 43
});
```

Here, the rejection handler is called as a result of p2 being rejected. The rejected value 43 from p2 is passed into that rejection handler.

Returning thenables from fulfillment or rejection handlers doesn't change when the promise executors are executed. The first defined promise will run its executor first, followed by the second, and so on. Returning thenables simply allows you to define additional responses. You defer the execution of fulfillment handlers by creating a new promise within a fulfillment handler. For example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    console.log(value);     // 42

    // create a new promise
    let p2 = new Promise(function(resolve, reject) {
        resolve(43);
    });

    return p2
}).then(function(value) {
    console.log(value);     // 43
});
```

In this example, a new promise is created within the fulfillment handler for p1. That means the second fulfillment handler will not be executed until after p2 has been fulfilled. This pattern is useful when you want to wait until a previous promise has been settled before triggering another promise.

Responding to Multiple Promises

Up to this point, each example in this chapter has dealt with responding to one promise at a time. There are times, however, when you'll want to monitor the progress of multiple promises in order to determine the next action. ECMAScript 6 provides two methods that monitor multiple promises: Promise.all() and Promise.race().

Promise.all()

The Promise.all() method accepts a single argument, which is an iterable (such as an array) of promises to monitor, and returns a promise that is resolved only when every promise in the iterable is resolved. The returned promise is fulfilled when every promise in the iterable is fulfilled, for example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.all([p1, p2, p3]);

p4.then(function(value) {
    console.log(Array.isArray(value));  // true
    console.log(value[0]);              // 42
    console.log(value[1]);              // 43
    console.log(value[2]);              // 44
});
```

Each of the promises in this example resolves with a number. The call to Promise.all() creates a new promise, p4, that ultimately is fulfilled when all of the promises are fulfilled. The result passed to the fulfillment handler for p4 is an array containing each resolved value: 42, 43, and 44. In this way, you can match promise results to the promises that resolved to them.

If any of the promises passed to Promise.all() is rejected, the returned promise is immediately rejected without waiting for the other promises to complete:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.all([p1, p2, p3]);

p4.catch(function(value) {
    console.log(Array.isArray(value))    // false
    console.log(value);                  // 43
});
```

In this example, p2 is rejected with a value of 43. The rejection handler for p4 is called immediately without waiting for either p1 or p3 to finish executing (they still finish executing, it's just that p4 doesn't wait). The rejection handler is passed 43 to reflect the rejection from p2. The rejection handler always receives a single value rather than an array, and the value is the rejection value from the promise that was rejected.

Promise.race()

The Promise.race() method provides a slightly different take on monitoring multiple promises. This method also accepts an iterable of promises to monitor and returns a promise, however, the returned promise is settled as soon as the first promise is settled. So instead of waiting for all promises to be fulfilled, as in Promise.all(), the returned promise is fulfilled as soon as any of the promises is fulfilled. For example:

```
let p1 = Promise.resolve(42);

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.race([p1, p2, p3]);

p4.then(function(value) {
    console.log(value);     // 42
});
```

In this code, p1 is created as a fulfilled promise while the others schedule jobs. The fulfillment handler for p4 is then called with the value of 42 and ignores the other promises completely. The promises passed to Promise.race() are truly in a race to see which is settled first. If the first promise to settle is fulfilled, then the returned promise is fulfilled; if the first promise to settle is rejected, then the returned promise is rejected. Here's an example with a rejection:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = Promise.reject(43);

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.race([p1, p2, p3]);

p4.catch(function(value) {
    console.log(value);     // 43
});
```

Here, p4 is rejected because p2 is already in the rejected state when Promise.race() is called. Even though p1 and p3 are fulfilled, those results are ignored because they occur after p2 is rejected.

Asynchronous Task Running

Back in chapter 8, you learned about generators and how they can be used for asynchronous task running such as the following:

```
let fs = require("fs");

function run(taskDef) {

    // create the iterator, make available elsewhere
    let task = taskDef();

    // start the task
    let result = task.next();

    // recursive function to keep calling next()
    function step() {

        // if there's more to do
        if (!result.done) {
            if (typeof result.value === "function") {
                result.value(function(err, data) {
                    if (err) {
                        result = task.throw(err);
                        return;
                    }

                    result = task.next(data);
                    step();
                });
            } else {
                result = task.next(result.value);
                step();
            }

        }
    }

    // start the process
    step();

}
```

```
// Define a function to use with the task runner

function readFile(filename) {
    return function(callback) {
        fs.readFile(filename, callback);
    };
}

// Run a task

run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

There are some pain points to this implementation. First, wrapping every function in a function that returns a function is a bit confusing (even this sentence was confusing). Second, there is no way to distinguish between a function return value intended to be a callback for the task runner and one that is not. With promises, you can greatly simplify and generalize this process by ensuring that each asynchronous operation returns a promise. That common interface means you can greatly simplify asynchronous code:

```
let fs = require("fs");

function run(taskDef) {

    // create the iterator
    let task = taskDef();

    // start the task
    let result = task.next();

    // recursive function to iterate through
    (function step() {

        // if there's more to do
        if (!result.done) {

            // resolve to a promise to make it easy
            let promise = Promise.resolve(result.value);
            promise.then(function(value) {
                result = task.next(value);
                step();
            }).catch(function(error) {
                result = task.throw(error);
                step();
            });
        }
    }());
}

// Define a function to use with the task runner

function readFile(filename) {
    return new Promise(function(resolve, reject) {
        fs.readFile(filename, function(err, contents) {
            if (err) {
                reject(err);
            } else {
                resolve(contents);
            }
        });
    });
}

// Run a task

run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

In this version of the code, a generic run() function is used to execute a generator. The run() function executes the generator to create an iterator, starts the task by calling task.next(), and then recursively calls step() until the iterator is complete. Inside of step(), if there's more work to do then result.done is false. At that point, result.value should be a promise, but Promise.resolve() is used just in case the function in question didn't return a promise (remember, Promise.resolve() will just pass through any promise that is passed in and will wrap any non-promise in a promise). Then, a fulfillment handler is added that retrieves the promise value and passes it back to the iterator. After that, result is assigned to the next yield result before calling step(). A rejection handler is also added and assumes any rejection results in an error object. That error object is passed back into the iterator using task.throw() and result is assigned to the next yield result if the error is caught in the task, and then step() is called to continue.

This same run() function can be used to run any generator that uses yield as a way to achieve asynchronous code without exposing promises (or callbacks) to the developer. In fact, any function may be used with any return value because the result is always converted into a promise. That means you can use both synchronous and asynchronous methods work correctly when called using yield, and you never have to check that the return value is a promise. The only concern is ensuring that asynchronous functions, such as readFile(), return a promise that correctly identifies its state. In the case of Node.js built-in methods, that means you'll have to convert those methods to return promises instead of using callbacks.

Inheriting from Promises

Just like other built-in types, promises can be used as a base upon which you can create a derived class. This allows you to define your own variation of promises to extend what the built-in promises can do. Suppose, for instance, you'd like to create a promise that uses success() and failure() in addition to then() and catch(). You could do so as follows:

```
class MyPromise extends Promise {

    // use default constructor

    success(resolve, reject) {
        return this.then(resolve, reject);
    }

    failure(reject) {
        return this.catch(reject);
    }

}

let promise = new MyPromise(function(resolve, reject) {
    resolve(42);
});

promise.success(function(value) {
    console.log(value);             // 42
}).failure(function(value) {
    console.log(value);
});
```

In this example, MyPromise is derived from Promise and two additional methods are added. Both success() and failure() use this to call the appropriate method they are mimicking. The created promise functions the same as the built-in version, except now you can use success() and failure() in addition to then() and catch().

Since static methods are also inherited, that means MyPromise.resolve(), MyPromise.reject(), MyPromise.race(), and MyPromise.all() are also present. While the last two behave the same as the built-in methods, the first two are slightly different.

Both MyPromise.resolve() and MyPromise.reject() will return an instance of MyPromise regardless of the value passed because it uses the @@species property (see Chapter 9) to determine the type of promise to return. So if a built-in promise is passed to either, it will be resolved or rejected and return a new MyPromise so you can assign fulfillment and rejection handlers. For example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = MyPromise.resolve(p1);
p2.success(function(value) {
    console.log(value);         // 42
});

console.log(p2 instanceof MyPromise);    // true
```

Here, p1 is a built-in promise that is passed to MyPromise.resolve(). The result, p2, is an instance of MyPromise where the resolved value from p1 is passed into the fulfillment handler.

If an instance of MyPromise is passed to MyPromise.resolve() or MyPromise.reject(), it will just be returned directly without being resolved. In all other ways these two methods behave the same as Promise.resolve() and Promise.reject().

Summary

Promises are designed to improve asynchronous programming in JavaScript. Whereas events and callbacks have several limitations, the permutations available via promises mean more control and composability over asynchronous operations. This is accomplished by scheduling jobs to be added to the JavaScript engine's job queue for execution later. A second job queue keeps track of promise fulfillment and rejection handlers to ensure proper execution.

Promises have three states: pending, fulfilled, and rejected. A promise starts out in a pending state and is either fulfilled (a success) or rejected (a failure). In either case, handlers can be added to be notified when a promise is settled. The then() method allows you to assign a fulfillment and rejection handler and the catch() method allows you to assign only a rejection handler.

You can chain promises together in a variety of ways and pass information between them. Each call to then() creates and returns a new promise that is resolved when the previous one is resolved. Such chains can be used to trigger responses to a series of asynchronous events. You can also use Promise.race() and Promise.all() to monitor the progress of multiple promises and respond accordingly.

Asynchronous task running is made easier using generators in addition to promises, as promises give a common interface that asynchronous operations can return. You can then use generators and the `yield` operator to wait for asynchronous responses and respond appropriately.

Most new web APIs are being built on top of promises, and you can expect many more to follow suit in the future.

Modules

One of the most error-prone and confusing aspects of JavaScript has long been the "shared everything" approach to loading code. Whereas other languages have concepts such as packages, JavaScript lagged behind, and everything defined in every file shared the same global scope. As web applications became more complex and the amount of JavaScript used grew, the "shared everything" approach began to show problems with naming collisions, security concerns, and more. One of the goals of ECMAScript 6 was to solve this problem and bring some order into JavaScript applications. That's where modules come in.

What are Modules?

Modules are JavaScript files that are loaded in a special mode (as opposed to scripts, which are loaded in the original way JavaScript worked). At the time of my writing, neither browsers nor Node.js have a way to natively load ECMAScript 6 modules, but both have indicated that there will need to be some sort of opt-in to do so. The reason this opt-in is necessary is because module files have very different semantics than non-module files:

1. Module code automatically runs in strict mode and there's no way to opt-out of strict mode.
2. Variables created in the top level of a module are not automatically added to the shared global scope. They exist only within the top-level scope of the module.
3. The value of `this` in the top level of a module is `undefined`.
4. Modules do not allow HTML-style comments within the code (a leftover feature from the early browser days).
5. Modules must export anything that should be available to code outside of the module.

These differences may seem small at first glance, however, they represent a significant change in how JavaScript code is loaded and evaluated.

Module JavaScript files are created just like any other JavaScript file: in a text editor and typically with the `.js` extension. The only difference during development is that you use some different syntax.

Basic Exporting and Importing

The export keyword is used to expose parts of published code to other modules. In the simplest case, you can place export in front of any variable, function, or class declaration to export it from the module. For example:

```
// export data
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;

// export function
export function sum(num1, num2) {
    return num1 + num1;
}

// export class
export class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }
}

// this function is private to the module
function subtract(num1, num2) {
    return num1 - num2;
}

// define a function
function multiply(num1, num2) {
    return num1 * num2;
}

// export later
export multiply;
```

There are a few things to notice in this example:

1. Every declaration is exactly the same as it would otherwise be without the export keyword.
2. Both function and class declarations require a name. You cannot export anonymous functions or classes using this syntax (unless using the default

keyword discussed later in this chapter)

3. You need not always export the declaration, you can also export references, as with multiply in this example.
4. Any variables, functions, or classes that are not explicitly exported remain private to the module. In this example, subtract() is not exported and is therefore not accessible from outside the module.

An important limitation of export is that it must be used in the top-level of the module. For instance, this is a syntax error:

```
if (flag) {
    export flag;    // syntax error
}
```

This example is a syntax error because export is inside of an if statement. Exports cannot be conditional or done dynamically in any way. Part of the benefit of module syntax is so the JavaScript engine can staticly determine what will be exported. As such, you can only use export at the top-level of a module.

If you are using a transpiler like Babel.js, you may find that export can be used anywhere. This only works when code is converted to ECMAScript 5 and will not work with a native ECMAScript 6 module system.

Once you have a module with exports, you can access the functionality in another module by using the import keyword. An import statement has two parts: the identifiers you're importing and the module from which those identifiers should be imported. The basic form is as follows:

```
import { identifier1, identifier2 } from "module";
```

The curly braces after import indicate the identifiers to import from the given module. The keyword from is used to indicate the module from which to import the given identifiers. The module is specified using a string. At the time of my writing, it is still undecided what module identifiers will look like. They may end up being full file paths (such as "../mymodule.js"), file paths without extensions (such as "../mymodule"), or something else. This likely won't be determined until browsers and Node.js begin implementing modules natively.

Even though it looks similar, the list of identifiers to import is not a destructured object.

When importing an identifier from a module, the identifier acts as if it were defined using const. That means you cannot define another variable with the same name, use the identifier prior to the import statement, or change its value.

Suppose that the first example in this section is in a module named "example". You can import and use identifiers from that module in a number of ways. You can just import one identifier:

```
// import just one
import { sum } from "example";

console.log(sum(1, 2));     // 3

sum = 1;        // error
```

This example imports only sum() from the example module. Even though the example module exports more than just that one function, they are not exposed here. If you try to assign a new value to sum, the result is an error, as you cannot reassign imported identifiers.

If you want to import multiple identifiers from the example module, you can explicitly list them out:

```
// import multiple
import { sum, multiply, magicNumber } from "example";
console.log(sum(1, magicNumber));   // 8
console.log(multiply(1, 2));        // 2
```

Here, three identifiers are imported from the example module: sum, multiply, and magicNumber. They are then used as if they were locally defined.

There’s also a special case that allows you to import the entire module as a single object. All of the exports are then available on that object as properties. For example:

```
// import everything
import * as example from "example";
console.log(example.sum(1,
        example.magicNumber));          // 8
console.log(example.multiply(1, 2));    // 2
```

In this code, the entirety of the example module is loaded into an object called example. The named exports sum(), multiple(), and magicNumber are then accessible as properties on example.

Keep in mind that the code inside of a module will only ever be executed once, regardless of the number of times it's used in an import statement. Consider the following:

```
import { sum } from "example";
import { multiply } from "example";
import { magicNumber } from "example";
```

Even though there are three import statements in this module, the code in "example" will only be executed once. The instantiated module is then kept in memory and reused whenever another import statement references it. It doesn't matter if the import statements are all in the module, or are spread across multiple modules - they each will use the same module instance.

Renaming Exports and Imports

Sometimes the original name of a variable, function, or class isn't what you want to use. It's possible to change the name of an export both during the export and when the identifier is being imported.

In the first case, suppose you have a function that you'd like to export with a different name. You can use the as keyword to specify the name that the function should be known as outside of the module:

```
function sum(num1, num2) {
    return num1 + num2;
}

export { sum as add };
```

Here, the sum() function (sum is the local name) is exported as add() (add is the exported name). That means when another module wants to import this function, it will have to use the name add instead:

```
import { add } from "example";
```

If the module importing the function wants to use a different name, it can also use as:

```
import { add as sum } from "example";
console.log(typeof add);            // "undefined"
console.log(sum(1, 2));             // 3
```

This code imports the add() function (the import name) and renames it to sum() (the local name). That means there is no identifier named add in this module.

Imported Bindings

A subtle but important point about the import statements is that they create bindings to variables, functions, and classes rather than simply referencing them. That means even though you cannot change an imported identifier, it can still change on its own. For example, suppose you have this module:

```
export var name = "Nicholas";
export function setName(newName) {
    name = newName;
}
```

When you import name and setName(), you can see that setName() is able to change the value of name:

```
import { name, setName } from "example";

console.log(name);       // "Nicholas"
setName("Greg");
console.log(name);       // "Greg"

name = "Nicholas";       // error
```

The call to setName("Greg") goes back into the module from which setName() was exported and executes there, setting name to "Greg". Note this change is automatically reflected on the imported name binding. That's because name is the local name for the exported name identifier so they are not the same thing.

Exporting and Importing Defaults

The module syntax is really optimized for exporting and importing default values from modules. The default value for a module is a single variable, function, or class as specified by the `default` keyword. For example:

```
export default function(num1, num2) {
    return num1 + num2;
}
```

This module exports a function as the default. The `default` keyword indicates that this is a default export and the function doesn't require a name because the module itself represents the function.

You can also specify an identifier as being the default export using the renaming syntax, such as:

```
// equivalent to previous example
function sum(num1, num2) {
    return num1 + num2;
}

export { sum as default };
```

The `as default` specifies that `sum` should be the default export of the module. This syntax is equivalent to the previous example.

⚠ You can only have one default export per module. It is a syntax error to use the `default` keyword with multiple exports.

You can import a default value from a module using the following syntax:

```
// import the default
import sum from "example";

console.log(sum(1, 2));     // 3
```

This import statement imports the default from the module `"example"`. Note that there are no curly braces used in this case, as would be with a non-default export. The local name `sum` is used to represent the function that the module exports. This syntax is the cleanest as it’s anticipated to be the dominant form of import on the web, allowing you to use already-existing object, such as:

```
import $ from "jquery";
```

For modules that export both a default and one or more non-defaults, you can import them with one statement. For instance, suppose you have this module:

```
export let color = "red";

export default function(num1, num2) {
    return num1 + num2;
}
```

You can then import both color and the default function using the following:

```
import sum, { color } from "example";

console.log(sum(1, 2));     // 3
console.log(color);         // "red"
```

The comma separates the default local name from the non-defaults (which are also surrounded by curly braces).

As with exporting defaults, importing defaults can also be accomplished using the renaming syntax:

```
// equivalent to previous example
import { default as sum, color } from "example";

console.log(sum(1, 2));     // 3
console.log(color);         // "red"
```

In this code, the default export (default) is renamed to sum and the additional color export is also imported. This example is equivalent to the previous example.

Re-exporting

There may be a time when you’d like to re-export something that your module has imported. You can do this using the patterns already discussed in this chapter, such as:

```
import { sum } from "example";
export { sum }
```

However, there’s also a single statement that can accomplish the same thing:

```
export { sum } from "example";
```

This form of export looks into the specified module for the declaration of sum and then exports it. Of course, you can also choose to export a different name for the same thing:

```
export { sum as add } from "example";
```

Here, sum is imported from "example" and then exported as add.

If you’d like to export everything from another module, you can use the * pattern:

```
export * from "example";
```

By exporting everything, you’re including the default as well as any named exports, which may affect what you can export from your module. For instance, if "example" has a default export, you’ll be unable to define a new default export when using this syntax.

Importing Without Bindings

Some modules may not export anything, and instead, only make modifications to objects in the global scope. Even though top-level variables, functions, and classes inside of modules do not automatically end up in the global scope, that doesn’t mean modules cannot access the global scope. The shared definitions of built-in objects such as Array and Object are accessible inside of a module and changes to those objects will be reflected in other modules.

For instance, suppose you want to add a method to all arrays called pushAll(), you may define a module like this:

```
// module code without exports or imports
Array.prototype.pushAll = function(items) {

    // items must be an array
    if (!Array.isArray(items)) {
        throw new TypeError("Argument must be an array.");
    }

    // use built-in push() and spread operator
    return this.push(...items);
};
```

This is a valid module even though there are no exports or imports. This code can be used both as a module and a script. Since it doesn't export anything, you can use a simplified import to execute the module code without importing any bindings:

```
import "example";

let colors = ["red", "green", "blue"];
let items = [];

items.pushAll(colors);
```

In this example, the module is imported and executed, so `pushAll()` is added to the array prototype. That means `pushAll()` is now available for use on all arrays inside of this module.

Imports without bindings are most likely to be used to create polyfills and shims.

Summary

ECMAScript 6 adds modules to the language as a way to package up and encapsulate functionality. Modules behave differently than scripts, as they do not modify the global scope with their top-level variables, functions, and classes, and `this` is `undefined`. In order to work differently than scripts, modules must be loaded using a different mode.

You must export any functionality you'd like to make available to consumers of a module. Variables, functions, and classes can all be exported, and there is also one default export allowed per module. After exporting, another module can import all or some of the exported names. These names act as if defined by let, and so operate as block bindings that cannot be redeclared in the same module.

Modules need not export anything if they are manipulating something in the global scope. In that case, it’s possible to import from such a module without introducing any bindings into the module scope.

Appendix A: Other Changes

Along with the changes already mentioned in the book, ECMAScript 6 has made some very small changes and improvements. This appendix lists out those changes.

Working with Integers

A lot of confusion has been caused over the years related to JavaScript’s single number type that is used to represent both integers and floats. The language goes through great pains to ensure that developers don’t need to worry about the details, but problems still leak through from time to time. ECMAScript 6 seeks to address this by making it easier to identify and work with integers.

Identifying Integers

The first addition is Number.isInteger(), which allows you to determine if a value represents an integer in JavaScript. Since integers and floats are stored differently, the JavaScript engine looks at the underlying representation of the value to make this determination. That means numbers that look like floats might actually be stored as integers and therefore return true from Number.isInteger(). 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

However, all is not so simple with integers. JavaScript can only accurately represent integers between -2^{53} and 2^{53}, and outside of this “safe” range, binary representations end up reused for multiple numeric values. For example:

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

This example doesn’t contain a typo, two different numbers end up represented by the same JavaScript integer. The effect becomes more prevalent the further the value is outside of the safe range.

ECMAScript 6 introduces `Number.isSafeInteger()` to better identify integers that can accurately be represented in the language. There is also `Number.MAX_SAFE_INTEGER` and `Number.MIN_SAFE_INTEGER` that represent the upper and lower bounds of the same range, respectively. The `Number.isSafeInteger()` method ensures that a value is an integer and falls within the safe range of integer values:

```
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 `Number.isInteger()` and `Number.isSafeInteger()`. The number `outside` is the first questionable integer value, so it is no longer 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 it’s a good idea to use `Number.isSafeInteger()` as part of input validation.

New Math Methods

The aforementioned new emphasis on gaming and graphics in JavaScript led to the realization that many mathematical calculations could be done more efficiently by a JavaScript engine than with pure JavaScript code. 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. It’s important, for instance, to know whether the numbers should be treated as 32-bit integers or as 64-bit floats.

As a result, ECMAScript 6 adds several new methods to the `Math` object. These new methods are important for improving the speed of common mathematical calculations, and therefore, improving the speed of applications that must perform many calculations (such as graphics programs). The new methods are listed below.

Method	Description
`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.
	Returns the result of performing true 32-bit

Method	Description
Math.imul(x, y)	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. However, if you are looking for a reasonably common calculation, be sure to check the new `Math` methods before implementing it yourself.

Unicode Identifiers

Better Unicode support in ECMAScript 6 also means changes to what characters may be used for an identifier. In ECMAScript 5, it was already possible to use Unicode escape sequences for identifiers, such as:

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

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

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

In ECMAScript 6, you can also use Unicode code point escape sequences as identifiers:

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

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

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

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

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` (zero-width non-joiner), `\u200d` (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 `[[Prototype]]`. Effectively, `__proto__` was an early precursor to both the `Object.getPrototypeOf()` and `Object.setPrototypeOf()` methods. It was unrealistic to expect all JavaScript engines to remove this property, so ECMAScript 6 also formalized the `__proto__` behavior. However, the formalization is in Appendix B 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.

While it's best to avoid using `__proto__`, it's interesting to see how the specification defined it. 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 the other methods, except that __proto__ allows you to set the prototype of an object literal directly. Here's how that works:

```
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
```

This example is functionally equivalent to the getGreeting() example. The call to Object.create() is replaced with an object literal that assigns a value to the __proto__ property. The only real difference between creating an object with Object.create() or an object literal with __proto__ is that the former requires you to specify full property descriptors for any additional object properties. The latter is just a standard object literal.

The `__proto__` property is special in a number of ways:

1. You can only specify it 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.

   For these reasons, it's recommend to avoid using `__proto__` when you can use `Object.getPrototypeOf()` and `Object.setPrototypeOf()` instead.