Introducing JavaScript Classes

Unlike most formal object-oriented programming languages, JavaScript didn't support classes and classical inheritance as the primary way of defining similar and related objects when it was created. This left many developers confused, and from pre-ECMAScript 1 all the way through ECMAScript 5, many libraries created utilities to make JavaScript look like it supports classes. While some JavaScript developers do feel strongly that the language doesn't need classes, the number of libraries created specifically for this purpose led to the inclusion of classes in ECMAScript 6.

While exploring ECMAScript 6 classes, it's helpful to understand the underlying mechanisms that classes use, so this chapter starts by discussing how ECMAScript 5 developers achieved class-like behavior. As you will see after that, 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

In ECMAScript 5 and earlier, JavaScript had no classes. The closest equivalent to a class was creating a constructor and then assigning methods to the constructor's prototype, an approach typically called creating a custom type. For example:

```js
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 an instance with a single property called `name`. The `sayName()` method is assigned to the prototype so the same function is shared by all instances of the `PersonType` object. Then, a new instance of `PersonType` is created via the `new` operator. The resulting `person` object is considered an instance of `PersonType` and of `Object` through prototypal inheritance.

This 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 in ECMAScript 6 is the class declaration, which looks similar to classes in other languages.

A Basic 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 a simple class declaration:

```
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. But instead of defining a function as the constructor, class declarations allow you to define the constructor directly inside the class with 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 methods as you want.

I> *Own properties*, properties that occur on the instance rather than the prototype, can only be created inside a class constructor or method. In this example, `name` is an own property. I recommend creating all possible own properties inside the constructor function so a single place in the class is responsible for all of them.

Interestingly, class declarations 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` gives `"function"` as the result. The `sayName()` method also ends up as a method on `PersonClass.prototype` in this example, similar to the relationship between `sayName()` and `PersonType.prototype` in the previous example. These similarities allow you to mix custom types and classes without worrying too much about which you're using.

Why to Use the Class Syntax

Despite the similarities between classes and custom types, 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 lack an internal `[[Construct]]` method and will 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 code, which doesn't use the class syntax:

```
// 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;
}());
```

First, notice that there are two `PersonType2` declarations: a `let` declaration in the outer scope and a `const` declaration inside the IIFE. This is how class methods are forbidden from overwriting the class name while code outside the class is allowed to do so. The constructor function checks `new.target` to ensure that it's being called with `new`; if not, an error is thrown. Next, the `sayName()` method is defined as nonenumerable, and the method checks `new.target` to ensure that it wasn't called with `new`. The final step returns the constructor function.

This example shows that while it's possible to do everything classes do without using the new syntax, the class syntax simplifies all of the functionality significantly.

A> ### Constant Class Names A> A> The name of a class is only specified as if using `const` 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: A> A> `js A> class Foo { A> constructor() { A> Foo = "bar"; // throws an error when executed A> } A> } A> A>// but this is okay after the class declaration A> Foo = "baz"; A>` A> A> In this code, the `Foo` inside the class constructor is a separate binding from the `Foo` outside the class. The internal `Foo` is defined as if it's a `const` and cannot be overwritten. An error is thrown when the constructor attempts to overwrite `Foo` with any value. But since the external `Foo` is defined as if it's a `let` declaration, you can overwrite its value 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.

A Basic Class Expression

Here's the class expression equivalent of the previous `PersonClass` examples, followed by some code that uses it:

```
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"
```

As this example demonstrates, class expressions do not require identifiers after `class`. Aside from the syntax, class expressions are functionally equivalent to class declarations.

Whether you use class declarations or class expressions is mostly 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 just like function expressions, you can also name class expressions. To do so, include an identifier after the `class` keyword like this:

```
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 named `PersonClass2`. The `PersonClass2` identifier exists only within the class definition so that it can be used inside the class methods (such as the `sayName()` method in this example). Outside the class, `typeof PersonClass2` is `"undefined"` because no `PersonClass2` binding exists there. To understand why this is, look at an equivalent declaration that doesn't use 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() {

            // 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 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`). A named class expression uses its name in the `const` definition, so `PersonClass2` is defined for use only inside the class.

While named class expressions behave differently from 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, which I'll cover next.

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 they're 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, the `createObject()` function is called with an anonymous class expression as an argument, creates an instance of that class with `new`, and returns the instance. The variable `obj` then stores the returned instance.

Another interesting use of class expressions is creating 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 {

    constructor(name) {
        this.name = name;
    }

    sayName() {
        console.log(this.name);
    }

}("Nicholas");

person.sayName();       // "Nicholas"
```

Here, an 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 of the class expression 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 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 code, 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` property on the element itself. This accessor property is created on the `CustomHTMLElement.prototype` and, just like any other method would be, is created as non-enumerable. 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 can save by using a class instead of the non-class equivalent. The `html` accessor property definition alone is almost the size of the equivalent class declaration.

Computed Member Names

The similarities between object literals and classes aren't quite over yet. Class methods and accessor properties can also have computed names. Instead of using an identifier, use square brackets around an expression, which is the same syntax used for 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 a name to a method inside its definition. The string `"sayName"` is assigned to the `methodName` variable, and then `methodName` is used to declare the method. The `sayName()` method is later accessed directly.

Accessor properties can use computed names in the same way, like this:

```
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 `propertyName` variable. Accessing the property by using `.html` only affects the definition.

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 Chapter 8 introduced generators, 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` with a `createIterator()` generator method. The method returns an iterator whose values are hardcoded into the generator. Generator methods are useful when you have an object that represents a collection of values and you'd like to iterate over those values easily. Arrays, sets, and maps all have multiple generator methods to account for the different ways developers need to interact with their items.

While generator methods are useful, defining a default iterator for your class is much more helpful if the class represents a collection of values. 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 the `this.items` array. Any class that manages a collection of values should include a default iterator because some collection-specific operations require collections they operate on to have an iterator. Now, any instance of `Collection` can be used directly in a `for-of` loop or with the spread operator.

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

Adding additional methods directly onto constructors to simulate static members is another common pattern in ECMAScript 5 and earlier. 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");
```

In other programming languages, the factory method called `PersonType.create()` would be considered a static method, as it doesn't depend on an instance of `PersonType` for its data. ECMAScript 6 classes simplify the creation of static members by using the formal `static` annotation before the method or accessor property name. For instance, here's the class 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 used for `sayName()` except for 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 can't use `static` with the `constructor` method definition.

W> Static members are not accessible from instances. You must always access static members from the class directly.

Inheritance with Derived Classes

Prior to ECMAScript 6, implementing inheritance with custom types was an extensive process. Proper inheritance required multiple steps. For instance, consider this example:

```
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: false,
        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
```

`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 the `Rectangle.call()` method. These steps often confused JavaScript newcomers and were a source of errors for experienced developers.

Classes make inheritance easier to implement 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 by calling the `super()` method. Here's the ECMAScript 6 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
```

This time, 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`).

Classes that inherit from other classes are referred to as *derived classes*. Derived classes require you to use `super()` 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.

W> There are a few things to keep in mind when using `super()`: W> W> 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. W> 1. 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. W> 1. 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` 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;
    }
}
```

Since `getArea()` is defined as part of `Square`, the `Rectangle.prototype.getArea()` method 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 the `super.getArea()` method, like this:

```
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 works the same as the the super references discussed in Chapter 4 (see "Easy Prototype Access With Super References"). 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. Inheritance works like that in other languages, but this 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 the `Rectangle` class. Through inheritance, that method is available as `Square.create()` and behaves in the same manner as the `Rectangle.create()` method.

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 instance:

```
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
```

`Rectangle` is defined as an ECMAScript 5-style constructor while `Square` is a class. Since `Rectangle` has `[[Construct]]` and a prototype, the `Square` class can still inherit directly from it.

Accepting any type of expression after `extends` offers 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
```

The `getBase()` function is called directly as part of the class declaration. It returns `Rectangle`, making this example is functionally equivalent to the previous one. And since you can determine the base class dynamically, 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. The `mixin()` function dynamically populates the prototype of a new function with all of the own properties of each mixin. (Keep in mind that if multiple mixins have the same property, only the last property added will remain.)

W> Any expression can be used after `extends`, but not all expressions result in a valid class. Specifically, the following expression types cause errors: W> W> * `null` W> * generator functions (covered in Chapter 8) W> W> 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 JavaScript arrays have existed, developers have wanted to create their own special array types through inheritance. 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"
```

The `console.log()` output at the end of this code shows how using the classical form of JavaScript inheritance on an array results in unexpected behavior. The `length` and numeric properties on an instance of `MyArray` don't behave the same as they do for the built-in array because this functionality isn't covered either by `Array.apply()` or by assigning the prototype.

One goal 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 (like the `Array.apply()` method) is called. 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 with all the built-in functionality of the base and correctly receives all functionality related to it.

The following example shows a class-based special array in action:

```
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
```

`MyArray` inherits directly from `Array` and therefore works like `Array`. Interacting with numeric properties updates the `length` property, and manipulating the `length` property updates the numeric properties. That means you can both properly inherit from `Array` to create your own derived array classes and inherit from other built-ins as well. With all this added functionality, ECMAScript 6 and derived classes have effectively removed the last special case of inheriting from built-ins, but that case is still worth exploring.

The Symbol.species Property

An interesting aspect of inheriting from built-ins is that any method that returns an instance of the built-in will automatically return a derived class instance instead. So, if you have a derived class called `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. However, the constructor for the return value is read from the `Symbol.species` property, allowing for this change.

The `Symbol.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). The following builtin types have `Symbol.species` defined:

- `Array`
- `ArrayBuffer` (discussed in Chapter 10)
- `Map`
- `Promise`
- `RegExp`
- `Set`
- Typed Arrays (discussed in Chapter 10)

Each of these types has a default `Symbol.species` property that returns `this`, meaning the property will always return the constructor function. If you were to implement that functionality 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 changing the species of a class isn't possible. 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, `MyDerivedClass1` inherits from `MyClass` and doesn't change the `Symbol.species` property. When `clone()` is called, it returns an instance of `MyDerivedClass1` because `this.constructor[Symbol.species]` returns `MyDerivedClass1`. The `MyDerivedClass2` class inherits from `MyClass` and overrides `Symbol.species` to return `MyClass`. When `clone()` is called on an instance of `MyDerivedClass2`, the return value is an instance of `MyClass`. Using `Symbol.species`, any derived class can determine what type of value should be returned when a method returns an instance.

For instance, `Array` uses `Symbol.species` to specify the class to use for methods that return an array. In a class derived from `Array`, you can determine the type of object returned from the inherited methods, 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 `Symbol.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 `Symbol.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 `Symbol.species` defined, be sure to use that value instead of the constructor.

Using new.target in Class Constructors

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
```

This code shows that `new.target` is equivalent to `Rectangle` when `new Rectangle(3, 4)` is called. Class constructors can't be called without `new`, so the `new.target` property is always defined inside of class constructors. But the value may not always be the same. Consider this code:

```
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
```

`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 can't be instantiated directly) by using `new.target` as follows:

```
// 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.

I> Since classes can't be called without `new`, the `new.target` property is never `undefined` inside of a class constructor.

Summary

ECMAScript 6 classes make inheritance in JavaScript easier to use, so you don't need to throw away any existing understanding of inheritance you might have from other languages. 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, a feature that better matches the behavior of built-in objects for which methods are typically non-enumerable by default. Additionally, class constructors can't 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 like `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 JavaScript. They provide a more concise syntax and better functionality for defining custom object types in a safe, consistent manner.