Extracted from:

Rediscovering JavaScript

Master ES6, ES7, and ES8

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Rediscovering JavaScript

Master ES6, ES7, and ES8

Venkat Subramaniam

edited by Jacquelyn Carter

Rediscovering JavaScript

Master ES6, ES7, and ES8

Venkat Subramaniam

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Copy Editor: Liz Welch
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

Printed in the United States of America.
ISBN-13: 978-1-68050-546-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2018

CHAPTER 7

Working with Classes

Older versions of JavaScript supported classes but without using the class keyword. In the past, you may have found object-oriented (OO) programming in JavaScript rather primitive, especially if you were familiar with other mainstream languages. Even today, many programmers still think that JavaScript has little to do with OO programming. One of the main reasons for that is that the old syntax and semantics of working with classes were very confusing and error-prone. Modern JavaScript puts an end to that misery; now it is possible to create beautiful OO code with JavaScript.

In this chapter you'll learn all about creating classes and defining both instance and static members. You'll quickly find that the new syntax is more intuitive, easier to write, easier to maintain, and less error prone than the old syntax. In addition to learning about classes, you'll explore the semantic differences from classes in other languages, how to examine properties, and how to make use of new built-in collection classes in JavaScript.

By the end of this chapter you will be able to not only freely use classes, but also mix the functional programming style from Chapter 5, *Arrow Functions and Functional Style*, on page ? with the OO style we focus on here.

Creating a Class

Classes are the most fundamental part of OO programming, and yet, in the earlier versions, JavaScript did not have an explicit keyword to define classes. It was never clear if we were working with a class or a function. Serious OO programming requires more rigorous syntax and a clearer specification for creating and using classes. Modern JavaScript delivers that quite well, fortunately.

We'll quickly revisit the classes of the past, which were masquerading as functions, so we can have a greater appreciation for the updated OO features that are now available in the language. Then we'll dive into the facilities to create classes—language capabilities that are bound to please the OO programmer in you.

Out with the Old Way

To create a class, we used to write a constructor function. The constructor looked much like any other function in syntax. To tell the difference between a regular function and a constructor, we relied on programmers following the convention to capitalize the function name. While function car() is considered a function, function Car() is considered a constructor. That's just the beginning of confusion. Let's take a look at how members of a class were defined.

It was not clear how to define the members of a class. Should we write

```
function Car() {
  this.turn = function(direction) { console.log('...turning...'); }
}
```

or

```
function Car() {}

Car.prototype.turn = function(direction) { console.log('...turning...'); }
```

or

```
function Car() {}

Car.turn = function(direction) { console.log('...turning...'); }
```

Each of these has consequences, and having different ways to define functions placed a burden on programmers and resulted in errors.

Another problem was that there was nothing to stop someone from placing a new before a function, like new car(), or invoking a constructor as a function, like Car(). Accidentally using a piece of code in ways other than it was intended is a source of error and a huge time sink.

What about inheritance? And how do we override a method? Do we use

```
this.__proto__.foo.apply(this, arguments);
```

to call the method of the base class? Coding that is a form of cruel and unusual punishment. Not only was the syntax unclear, the approach was verbose and highly error prone.

Enough of that—out with the horrific old, in with the new, enriched, pleasant syntax.

In with the New Way

JavaScript has streamlined the syntax to create classes. The keyword class makes the intent obvious and unambiguous—you don't have to wonder anymore if the programmer meant a function or a constructor. Let's define a class named Car:

classes/define-class.js

```
class Car {}

console.log(Car);
```

The keyword class followed by an optional name of the class and an empty {} is the minimum syntax to define a class. It's that simple—no fuss, no confusion.

Even though we used the class syntax, we're actually creating a function, one that can only be used with new. The output from the code shows this:

```
[Function: Car]
```

In spite of the fact that the class syntax defines a function—albeit reserved for creating an object—if we invoke the class as if it were a regular function, we'll get an error:

```
Car(); //BROKEN CODE
^

TypeError: Class constructor Car cannot be invoked without 'new'
```

Furthermore, unlike the function Car() syntax, class Car does not hoist the definition of Car—that is, the definition of the class is not moved to the top of the file or function. The class is available for use only after the point of definition in the execution flow. Thus, the following code is not valid:

classes/no-hoisting.js

```
new Car(); //BROKEN CODE

class Car {}
```

If we refer to Car before we define it, we'll get an error, like so:

```
ReferenceError: Car is not defined
```

However, if the definition comes to life before the point of reference in the flow of execution, as in the following example, then it's all good:

classes/ok-use.js

```
const createCar = function() {
  return new Car();
};

class Car {}

console.log(createCar());
```

In short, the new syntax makes defining a class a pleasant and effortless experience, removes the issues with incorrect use, and at the same time, keeps the semantics of defining a class the same as before.

Implementing a Constructor

You know how to create a class, but you haven't seen how to define the body of the constructor. Creating a class defines a no-parameter default constructor, which appears to be empty bodied. But you may want to execute some code as part of object construction. For that we need to define an explicit constructor. That's exactly what we'll do next.

Let's first examine the default constructor that is automatically created when you define a class.

classes/default-constructor.js
```
class Car {}

console.log(Reflect.ownKeys(Car.prototype));
```

We created a class named Car, with a default constructor. Then, using the Reflect class's ownKeys() method, we examine the properties of the Car's prototype—you'll learn about Reflect in Chapter 12, *Deep Dive into Metaprogramming*, on page ?. This reveals the default constructor that JavaScript quietly created for us:

```
[ 'constructor' ]
```

We may provide an implementation or body for the constructor if we like. For that, we'll implement a special method named constructor in the class, like so:

classes/constructor.js
```
class Car {
  constructor(year) {
    this.year = year;
  }
}

console.log(new Car(2018));
```

The constructor may take zero, one, two, or any number of parameters, including default and rest parameters. The body of the constructor may initialize fields, like this.year in the example, and may perform actions. The output of the previous code shows that the constructor initialized the this.year field:

```
Car { year: 2018 }
```

A constructor is called when an instance is created using the new keyword. The constructor can't be called directly without new, as we saw earlier. If you do not have anything useful to do when an object is created, then do not implement a constructor—the default constructor is sufficient. If you want to initialize some fields or perform some actions when an instance is created, then write a constructor. However, keep the constructor short and the execution fast—you don't want to slow down during creation of objects.