Table of Contents

1
Important Concepts in JavaScript

Our application is going to be written in JavaScript, but there are many versions of JavaScript. These versions are formalized by **Ecma International** (formerly the **European Computer Manufacturers Association** (**ECMA**)), and are actually called **ECMAScript**. So the term "JavaScript" is a collective term for all of these different ECMAScript versions. Below, you'll find a table enumerating each version, alongside its release date:

ECMAScript version	Release year
1	1997
2	1998
3	1999
4	(never released)
5	2009
6	2015
7	2016
8	2017
9	2018

In other words, from 1999 to 2009, when developers were writing JavaScript they were actually writing in ECMAScript 3.

JavaScript was originally developed by Brendan Eich in 10 days! He wrote it for the Netscape Navigator browser. It was originally called *Mocha*, and this was changed to *LiveScript*, before finally settling on JavaScript. In 1996, Netscape submitted JavaScript to ECMA to be standardized as ECMAScript.

Other companies created similar languages; Microsoft created *VBSCript* and *JScript*, while Macromedia (now Adobe) created

ActionScript.

In this book, we will be using features up to **ECMAScript 2018** (a.k.a. **ES9**). These newer standards introduced new concepts, such as **classes**, and provided a cleaner syntax, such as **arrow functions**. Many developers use these features without understanding what they are or how they work. This chapter will explain in depth some of the most important concepts in JavaScript.

We will focus on three topics:

- Clarifying different **data types** in JavaScript.
- How **inheritance** works in JavaScript.
- Determining the current **context** in any piece of code.

Clarifying data types

In JavaScript, there are six **primitive data types** and one object type. The six primitive types are `null`, `undefined`, `Boolean` (`true`/`false`), `number`, `string`, and `symbol`. The object type is simply a key-value store:

```
const object = {
  key: "value"
}
```

To look at this from another angle, everything that is not a primitive type is an object. This means functions and arrays are both special types of object:

```
// Primitive types
true instanceof Object; // false
null instanceof Object; // false
undefined instanceof Object; // false
0 instanceof Object; // false
'bar' instanceof Object; // false
Symbol('foo') instanceof Object; // false

// Non-primitive types
(function () {}) instanceof Object; // true
[] instanceof Object; // true
({}) instanceof Object // true
```

If you are not familiar with JavaScript's data types, take a look at the MDN Web Docs on JavaScript data types and data structures (`developer.mozilla.org/en-`

`US/docs/Web/JavaScript/Data_structures`).

Primitives versus objects

So, what are the differences between primitives and objects? There are three main differences:

- Primitives are stored and compared *by value*; objects are stored and compared *by reference*:

```
42 === 42; // true
"foo" === "foo"; // true

{} === {}; // false
[] === []; // false
(function () {}) === (function () {}); // false
```

  Here, the value of the number `42` is simply `42`. However, the value of `{}` is actually a reference to the object, not the object itself. When we create two empty object literals, they are considered *as two different objects*, both with their own unique reference.

- Primitives cannot not have methods or properties, whereas objects can:

```
const answer = 42
answer.foo = "bar";
answer.foo; // undefined
```

- As an extension of these points, objects are **mutable**, so you can add, change, or remove properties and methods from them.

```
const foo = {};
foo.bar = "baz";
foo; // { bar: "baz" }
```

  On the contrary, primitives are immutable; you cannot change the number `42` to anything else without it being a different number. When you do perform an operation on a primitive (multiplying a number or adding characters to a string), a new primitive value is created; the original value remains unchanged.

Under the hood, when you instantiate a new primitive, a section of memory is allocated to the primitive. For example, when we define the string `"foo"`, a small portion of the memory with the address `A` is allocated to store the string. When we

perform an operation on it, appending `"bar"` to it, for example, a new string, `"foobar"`, is created with a different address, `B`. The original `"foo"` is not altered.

Functions

A function is not a type of primitive; the rules outlined above make it an object—a special type of object with special properties and methods such as `name` and `call`:

```
const foo = function bar(baz) { console.log(baz); };
foo.name; // "bar"
foo.call(this, "Hello World!"); // "Hello World!"
```

A method is an object property that also happens to be a function:

```
const car = {};

// honk is a method of car
car.honk = function () { console.log("HONK!"); };
car.honk(); // "HONK!"
```

Just like other objects, you can add new properties and methods to it:

```
foo.qux = "baz";
foo.qux; // "baz"
```

Because functions are objects, they are treated as **first-class citizens**. This means functions can be assigned to variables, and both can be passed in as arguments to, and returned from, another function:

```
// Function assigned to a variable
const requestHandler = function (request, response) {
    console.log(request);
    response.end("Hello World!");
}

// Passing in a function as argument to another function
const server = http.createServer(requestHandler);
```

Constructor functions

In your program, you'll often need to create multiple objects that have the same properties and methods. You can do this by defining a **constructor function** and invoking it with the `new` keyword:

```
const Car = function (make) {
 this.make = make;
 this.honk = function () { console.log("HONK!") }
}

const myCar = new Car("BMW");
myCar.make; // "BMW"
myCar.honk(); // "HONK!"
```

The constructor function contains information on how to construct an object. Any function (except arrow functions) can be a constructor function; a function is only used as a constructor function when called with the `new` keyword:

```
const Car = function () {}
const myCar = new Car();
myCar; // {}
```

Here, an empty function is used as a constructor function by invoking it with the `new` keyword. Since there's no logic inside the function body, it simply returns an empty object.

Within the function block, `this` refers to the object that's to be created, so we can assign properties to the object by adding them to `this`:

```
const Car = function (make) {
  this.make = make;
}
const myCar = new Car("BMW");
myCar; // { make: "BMW" }

const yourCar = new Car("Audi");
yourCar; // { make: "Audi" }
```

Just to demonstrate a point, when you run a function normally (without the `new` keyword), it will act like any other function:

```
const Car = function (make) {
  this.make = make;
}
const myCar = Car("BMW");
myCar; // undefined
window.make; // "BMW"
```

Since our function did not return anything, JavaScript implicitly returns `undefined`. Also, since our function is run on the top-level code, the current context, as represented by `this`, is the *global context*, which, in browsers, is the `window` object.

Wrapper objects and autoboxing

You may not directly use constructor functions a lot, but they are more prevalent than you may at first realize. For instance, they are the reason that the following operation works:

```
const answer = 42;
answer.toString(); // "42"
```

Since primitives cannot have properties and methods, how can we call the `toString` method of `answer`? It turns out that some primitive types, namely Booleans, strings, and numbers, have an object equivalent.

```
const stringPrimitive = "foo";
const stringObject = new String("foo");

stringPrimitive === "foo"; // true
stringObject === "foo"; // false

stringPrimitive instanceof Object; // false
stringObject instanceof Object; // true
```

That new object is often referred to as a **wrapper object**. For example, our `stringObject` would have the following structure:

```
{
  0: "f",
  1: "o",
  2: "o",
  length: 3,
  __proto__: String,
  [[PrimitiveValue]]: "foo"
}
```

When we try to access the `toString` method of the primitive, JavaScript automatically, and *temporarily*, wraps our primitive with its corresponding wrapper function and accesses the method from there. Since every object inherits a `toString` method, we are able to call `toString` on our wrapper object. This automatic wrapping is known as **autoboxing**.

Autoboxing is a temporary process. After the `toString` method has returned, the wrapper object is discarded and the primitive is unchanged.

This also explains why no errors are thrown when you try to assign a property to a primitive. This is because the assignment is not actually done on the primitive, but on the temporary wrapper object.

```
const foo = 42;
foo.bar = "baz"; // Assignment done on temporary wrapper object
foo.bar; // undefined, since wrapper object is discarded after
assignment
```

Creating objects

As I mentioned already in `Chapter 1`, *The Importance of Good Code*, programming is a creative endeavor; there are many ways to achieve the same results. This is also true for basic operations, such as creating new objects.

Factory pattern

Using a constructor function is not the only way to create new objects; the simplest way is to define a new object literal each time.

```
const myCar = {
  make: "BMW",
  honk: function () { console.log("HONK!") }
}
const yourCar = {
  make: "Audi",
  honk: function () { console.log("HONK!") }
}
```

However, this does not follow the **Don't Repeat Yourself** (**DRY**) principle. Instead, we can use the **factory pattern** and create a function that returns a new object each time it is called.

For example, here's how you can use the factory pattern to create the same `Car` object.

```
const createCar = function (make) {
  return {
    make: make,
    honk: function () { console.log("HONK!") }
  }
}

const myCar = createCar("BMW");
const yourCar = createCar("Audi");
```

Factory versus constructor

We can use either a factory or a constructor function to create new objects. So, when should you pick one over the other?

The benefit of using the constructor function is that you can define common methods to the function's `prototype` object (we'll cover more on prototypes later), which means all objects created using the constructor function would have shared access to that one function.

```
const Car = function (make) {
 this.make = make;
}

Car.prototype.honk = function () { console.log("HONK!") };

const myCar = new Car("BMW");
const yourCar = new Car("Audi");
myCar.honk === yourCar.honk; // true
```

With a factory, the object's method is duplicated each time a new object is created, which means it uses more memory.

```
const Car = function (make) {
 this.make = make;
 this.honk = function () { console.log("HONK!") }
}

const myCar = new Car("BMW");
const yourCar = new Car("Audi");
myCar.honk === yourCar.honk; // false
```

This answers one question and raises many more:

- What is the `prototype` object?
- How does the `new` keyword work?

To answer these questions, we must take a step back and understand the inheritance model of JavaScript.

Prototypes

JavaScript is a *multi-paradigm* language that can be used in an **object-oriented (OO)**, **functional**, or simply **procedural** way. The most common use case is OO. However,

unlike most OO languages, its inheritance model is not class-based, but prototype-based. Let's explore the differences.

Class-based inheritance

Class-based inheritance can be seen in the more 'traditional' OO languages, such as Java. In class-based inheritance, every object *must* belong to a class, and objects of a class *must* have all the methods and properties defined in the class, and nothing else. If you need an extra method or property, you'll need to define a new subclass and instantiate your object from that subclass.

Prototypical inheritance

To create an object using the prototype pattern, first define a prototype object that acts as the template, and then copy/clone that prototype object every time you want to create a new object of that type.

However, after you've created a new object from the prototype, you can add, remove, or modify any of the methods and properties of the object; the restrictions that class-based inheritance has do not apply here. The prototype acts only as an initial template, not a set of rules that objects must conform to at all times.

> *"In a prototype-based system, there are no classes. All objects are created by adding properties and methods to an empty object or by cloning an existing one."*
>
> *– from the article Javascript object prototype by Helen Emerson*
>
> `http://helephant.com/2009/01/18/javascript-object-prototype/`

Confusion arises because ECMAScript 2015 introduced the `class` keyword to the language, leading many to think JavaScript now supports classes in the classical sense. This is inaccurate; there are no classes in JavaScript and the `class` keyword is simply **syntactic sugar** for constructor functions. And, as you've seen before in our `Car` example, it is the constructor function that defines the template. Let's delve a little deeper to see how that works.

Prototype

Let's revisit our `Car` example:

```
const Car = function (make) {
```

```
 this.make = make;
}

Car.prototype.honk = function () { console.log("HONK!") };

const myCar = new Car("BMW");
myCar.make; // "BMW"
myCar.honk(); // "HONK!"
```

In JavaScript, every function has a `prototype` property.

Remember, since everything *except* primitive types is an object type, functions are objects and can have methods and properties.

When used as a constructor function, it is this `prototype` property, as well as the function body, that together provide the blueprint for the new object.

As mentioned previously, when used as a constructor function, the `this` keyword inside the function body refers to the object that is being constructed. So, when we assign the `make` variable to the `make` property of `this`, we are giving the object *itself* a `make` property:

```
const Car = function (make) {
 this.make = make;
}
const myCar = new Car("BMW");
myCar.make; // "BMW"
myCar.hasOwnProperty("make"); // true
```

Any objects constructed using the constructor function will have a *reference* to the constructor function's `prototype` property. Methods and properties defined in the constructor function's `prototype` can be accessed by the object, but are not properties of the object itself.

```
const Car = function () {}
Car.prototype.honk = function () { console.log("HONK!") };
const myCar = new Car("BMW");
myCar.honk(); // "HONK!"
myCar.hasOwnProperty("honk"); // false
```

So, how is that reference stored in the object?

[[Prototype]], proto, and prototype

Before we move forward, we must first be aware that there are three prototype-related constructs in JavaScript: `[[Prototype]]`, `__proto__`, and `prototype`. In the following section, we will elucidate each of their roles.

[[Prototype]]

In JavaScript, every object has an internal `[[Prototype]]` property, and it turns out the reference to the constructor function's `prototype` object is stored as this *hidden* `[[Prototype]]` property.

Conceptually, we can test this out as follows:

```
myCar.[[Prototype]] === Car.prototype; // technically `true`
```

However, this would throw a syntax error because `[[Prototype]]` is a hidden property and cannot be accessed directly. ECMAScript 5 provides the `Object.getPrototypeOf` method, which allows you access this hidden property:

```
Object.getPrototypeOf(myCar) === Car.prototype
```

__proto__

Prior to the standardization of `getPrototypeOf`, many browsers had already implemented a way to let you access the `[[Prototype]]` object. They added an **accessor property** (one which consists of a getter and setter), `__proto__`:

```
Object.getPrototypeOf(Car) === Car.__proto__; // true
Object.getPrototypeOf(myCar) === myCar.__proto__; // true
```

The `__proto__` property was not originally in the ECMAScript specification, but because many browsers have implemented it, the standardization body added it to the ECMAScript 2015 specification for backward compatibility.

From here on, we will refer to the `__proto__` property, since you can run the examples in your browser console, but remember that this is just one way to access the internal `[[Prototype]]` object.

The `__proto__` property references the `prototype` of the constructor function that constructed it; or, if the object was not created from a constructor function, it'll

reference the constructor function of its wrapper object:

```
const Car = function () {};
const myCar = new Car();
myCar.__proto__ === Car.prototype; // true

const foo = {}; // equivalent to new Object()
foo.__proto__ === Object.prototype; // true

const bar = []; // equivalent to new Array()
bar.__proto__ === Array.prototype; // true

const baz = function () {}; // equivalent to new Function()
baz.__proto__ === Function.prototype; // true
```

Quick recap

Just to recap, here are the main points:

- Objects can be created by invoking constructor functions with the `new` keyword
- Every function, including constructor functions, has a `prototype` property
- When an object is created using a constructor function, its properties are set inside the constructor function's body by modifying `this`
- When an object is created using a constructor function, a reference to the constructor function's `prototype` property is also stored in the object's internal `[[Prototype]]` property
- The object's `[[Prototype]]` property is hidden and cannot be accessed directly, so some browsers have implemented an accessor property, `__proto__`, which exposes the `[[Prototype]]` property

Prototype inheritance chain

Let's revisit our `Car` example one more time:

```
const Car = function (make) {
 this.make = make;
}

Car.prototype.honk = function () { console.log("HONK!") };

const myCar = new Car("BMW");
```

```
myCar.make; // "BMW"
myCar.honk(); // "HONK!"
```

In the preceding example, `honk` is not a method of the `myCar` object itself.

```
myCar.hasOwnProperty("honk"); // false
```

As we've already discussed, `honk` is a property of `Car.prototype`, which is referenced in the object's `__proto__` property.

```
{
  make: "BMW",
  __proto__: {
    honk: function () { console.log("HONK!") },
    constructor: Car
 }
}
```

In JavaScript, when you try to access a method or property, the engine will look at whether the object itself has that property/method; if it does, it'll use that one. However, if the property/method is not found, it will look inside the `__proto__` object and see whether it exists there; if it does, it'll use that one.

To demonstrate our point, let's add a `make` method to `Car.prototype`.

```
Car.prototype.make = function() { return "Audi"; }
myCar.make; // "BMW"
myCar.make === Car.prototype.make; // false
```

It still says `BMW` because the property in the object itself has priority over the one in the `myCar.__proto__` object. When we call `myCar.honk`, however, because `honk` is not in the object itself, it will use the one provided in `myCar.__proto__`.

```
myCar.honk === myCar.__proto__.honk; // true
myCar.honk === Car.prototype.honk; // true
```

As the last piece of the puzzle, we'll explain how we are able to call `myCar.hasOwnProperty`, since `hasOwnProperty` is neither a method on the object itself, nor a method of the object constructor's `prototype` object.

In fact, it is the `prototype` object of the object's constructor's constructor, which happens to be the `Object` constructor. After not finding the `hasOwnProperty` method in `myCar.__proto__` (that is, `Car.prototype`), it'll follow the same logic as before and try to find it in the `__proto__` of that object (remember, every object has a `__proto__` property).

```
{
  make: "BMW",
  __proto__: {
    honk: function () { console.log("HONK!") },
    constructor: Car,
    __proto__: {
      constructor: Object,
      hasOwnProperty: function () { ... }
      isPrototypeOf: function () { ... }
      ...
    }
  }
}
```

Because `Car.prototype` is an object, it can be thought of as being constructed using the `Object` wrapper method, and thus its `__proto__` property will be a reference to `Object.prototype`.

```
Car.prototype.__proto__ === Object.prototype; // true
myCar.__proto__.__proto__ === Object.prototype; // true
```

Therefore, when accessing a property or calling a method on an object, the JavaScript engine follows these rules:

- Try to find the property/method on the object:
  - If it's found, use it
  - If it's not found, look inside the `__proto__` object
- Repeat the first step until it is found; if it's not found, return `undefined`

The way inheritance is structured here is called the **prototype inheritance chain**; the JavaScript engine will follow the `__proto__` chain until it finds the property/method it is looking for.

Object property or prototype

So, we can give properties/methods to objects by adding them to the objects themselves, or we can add them to an object constructor's `prototype` object. So, which method should you choose?

This is a similar question to the one discussed in the *Factory versus constructor* section. If you add the method to the constructor's `prototype`, then all objects constructed from the constructor function will have access to the method. Because the object references its constructor's `prototype`, the method would still be available to objects

constructed before the method was added.

```
const Car = function () {};

const oldCar = new Car();
oldCar.turboDrive(); // TypeError: oldCar.turboDrive is not a function

Car.prototype.turboDrive = function () { console.log("TURBO Drive
activated!"); };

const newCar = new Car();
newCar.turboDrive(); // "TURBO Drive activated!"
oldCar.turboDrive(); // "TURBO Drive activated!"
```

Furthermore, only one method is stored in memory, instead of having to reserve space for each duplicated method defined on each object.

Therefore, if you have methods or properties that are not specific to an object instance, then you should define them in the prototype object of the constructor; on the other hand, if you have properties/methods whose values are specific to the object itself (taking the example of a car, this would be the speed and location), define them in the object itself.

Be careful when extending or modifying the `prototype` of "native" constructor functions such as `Object`, `Function`, `Array`, and so on. It will be inherited by *all* objects that use it, including any libraries or frameworks you're using. Furthermore, if you're overriding an existing method, it might inadvertently break the functionality of some libraries or frameworks that depend on it.

ECMAScript classes

The ECMAScript 2015 specification introduced the `class` keyword, which is syntactic sugar for constructor functions, making the syntax clearer and less bloated.

ECMAScript 2015 did not change how inheritance works in JavaScript; it only provided `class` as syntactic sugar.

Let's see how we can implement our `Car` constructor function as a class.

```
class Car {
```

```
  constructor(make) {
    this.make = make;
  }
  honk() { console.log("HONK!") };
}

const myCar = new Car("BMW");
myCar.make; // "BMW"
myCar.honk(); // "HONK!"
```

The `constructor` method replaced the constructor function's body, and **prototype methods** replaced the previous methods on the constructor function's `prototype`.

Methods

There are three types of method in a JavaScript class:

- Constructor
- Prototype
- Static

Constructor

`constructor` is a special method that gets called each time a new object is instantiated. The arguments passed in when creating a new object are passed on to the constructor, as seen in the following example:

```
class Bar {
  constructor(x) {
    console.log(x);
  }
}
const foo = new Bar("baz"); // "baz"
```

There must be one, and only one, constructor in a class. If you do not explicitly specify a constructor, the default one will be used, as seen here:

```
// Default constructor for non-derived classes
constructor() {}

// Default constructor for classes extended from a parent class
// (See 'extending a class' below)
constructor(...args) {
  super(...args);
```

```
}
```

Like the function body of the function constructor, `this` inside the method body refers to the new object that's being constructed, as seen in the following snippet:

```
class Car {
  constructor(make) {
    this.make = make;
  }
}

const myCar = new Car("BMW");
myCar.make; // "BMW"
```

The ES5 equivalent would be as follows:

```
const Car = function (make) {
  this.make = make;
}

const myCar = new Car("BMW");
myCar.make; // "BMW"
```

Prototype methods

Prototype methods are defined inside the class and are available to be called by their instances, as follows:

```
class Car {
  constructor(make) {
   this.make = make;
  }
  printMake() { console.log("I am a " + this.make + " car!") };
}

const myCar = new Car("BMW");
myCar.make; // "BMW"
myCar.printMake(); // "I am a BMW car!"
```

Within the body of a prototype method, `this` refers to the object instance, and so prototype methods can access the object's properties, like we did with `this.make`.

Prototypes are used for logic that is common to multiple instances *and* which requires access to the object instance.

The ES5 equivalent would be as follows:

```
const Car = function (make) {
  this.make = make;
}

Car.prototype.printMake = function () {
  console.log("I am a " + this.make + " car!");
};
```

Static methods

Static methods are methods that exist regardless of whether an instance exists or not. Therefore, they do not have access to instances' properties. Within the body of a static method, `this` refers to the class itself.

Static methods are callable on the class itself, but not on any of the instances, as seen in the following example:

```
class Foo {
  static bar() { console.log(this) } // Static method
  baz() { console.log(this) } // Prototype method
}

Foo.bar(); // class Foo { ... }, the class Foo itself
Foo.baz(); // Uncaught TypeError: Foo.baz is not a function

const qux = new Foo();
qux.bar() // Uncaught TypeError: baz.bar is not a function
qux.baz() // Foo {}, an instance of Foo
```

Static methods are used for sharing common functionalities relating to the class, such as utility functions, as follows:

```
class Car {
  static printVehicleType() { console.log("car") };
}

Car.printVehicleType(); // "car"
```

The ES5 equivalent would be as follows:

```
const Car = function () {};
Car.printVehicleType = function () { console.log("car") };
Car.printVehicleType(); // "car"
```

Extending a class

Along with the `class` keyword, ECMAScript 2015 also introduced the `extend` keyword, which allows you to create child classes that inherit from a parent class. The child class inherits both the prototype and static methods.

If the child class does not define a `constructor` method, the parent's `constructor` will be used. If the child class does define its own constructor, it must call `super()` at the start of the method so the parent's constructor is called first, as seen here:

```
class Vehicle {
  constructor() {
    this.type = "Vehicle";
  }
  printType() { console.log("type: " + this.type); }
  static kmToMile(km) { return km * 0.62137; }
}

class Car extends Vehicle {
  constructor() {
    super();
    this.type = "Car";
  }
}

Car.kmToMile(1); // 0.62137

const myCar = new Car();
myCar.printType(); // "Car"
```

`super` here refers to the **superclass**, another word for the parent class. Likewise, a child class may also be referred to as a **subclass**.

You can also extend constructor functions in ES5, but the syntax is more bloated and less elegant. To re-implement the previous two classes, it would look like this:

```
const Vehicle = function () {
  this.type = "Vehicle";
}
Vehicle.prototype.printType = function () {
  console.log("type: " + this.type);
}
Vehicle.kmToMile = function (km) {
  return km * 0.62137;
```

```
}

const Car = function () {
  Vehicle.call(this); // Same as Vehicle(), but passing through the
current context
  this.type = "Car";
}

// Creates an empty object who inherits from Vehicle
// So Car.prototype.__proto__ === Vehicle.prototype
Car.prototype = Object.create(Vehicle.prototype);

Car.prototype.constructor = Vehicle;
```

this and the context

In JavaScript, the `this` keyword refers to the current *context*, which is a reference to the object that owns the code being executed.

Global context

When used on top-level code, `this` refers to the **global context**, which is the `window` object when in a web browser, and the `exports` object inside a Node.js module.

```
this; // window
this.foo = "bar";
window.foo; // "bar"
```

Function context

When used inside a function, the value of `this` depends on *how* the function is called.

If it's a function called on top-level code, then the value of `this` remains the global context, as seen here:

```
const foo = function () { console.log(this) };
foo(); // window
```

A function block does not affect the context of its body.

```
const bar = function () { foo() };
bar(); // window
```

Note, however, if **strict mode** is on, then `this` becomes `undefined` inside the function block, as can be seen here:

```
const foo = function () { console.log(this) };
const bar = function () { 'use strict'; console.log(this); };

foo(); // window
bar(); // undefined
```

Object method

If the function is a method of an object, then `this` becomes the object that the method belongs to, as can be seen in the following example:

```
const func = function () { console.log(this) };
const obj = {};
obj.method = func;

obj.method(); // obj
```

Since a static method of a class is also just a normal object method, `this` refers to the class object inside the static method, as seen here:

```
class Foo {
  static staticMethod() {
    console.log(this);
  }
}
Foo.staticMethod(); // Foo
```

This also explains why when you call something like `window.setTimeout` inside another method, `this` becomes the `window` object inside the `setTimeout` method body, as seen here:

```
const foo = {};
foo.bar = function () {
  console.log(this); // foo
  window.setTimeout(function () {
    console.log(this); // window
  }, 0);
};
foo.bar();
```

Constructor functions

As we demonstrated in the previous section, when a function is used as a constructor function by calling it with the `new` keyword, `this` becomes the object that's constructed inside the function body:

```
function Car() {
  console.log(this);
}

const myCar = new Car(); // myCar
```

Since the `constructor` method inside a class is simply syntactic sugar for the body of the constructor function, `this` also represents the object being constructed inside the `constructor` method, as seen here:

```
class Car {
  constructor() {
    console.log(this);
  }
}

const myCar = new Car(); // myCar
```

Prototype methods

Inside a prototype method of a class, `this` refers to the instance of that class, as seen in the following example:

```
const Foo = function () {};
Foo.prototype.prototypeMethod = function () {
  console.log(this);
}

class Bar {
  prototypeMethod() {
    console.log(this);
  }
}

const foo = new Foo();
foo.prototypeMethod(); // foo
const bar = new Bar();
bar.prototypeMethod(); // bar
```

Binding this

As demonstrated, the value of `this` inside a function depends on how the function is called. However, there are times when you want to ensure that `this` refers to a particular value, *regardless* of how it is called. In these situations, you'd need to *bind* the value of `this`.

JavaScript provides the `bind` method, which is available for all functions. It returns a *new* function, with the `this` value bound to whatever you pass into the `bind` method, as shown here:

```
const foo = function () { console.log(this) };
const boundFoo = foo.bind(42);

foo(); // window
boundFoo(); // 42
```

`bind` is available because it inherits from `Function.prototype`:

```
const foo = function () {}; // Sames as new Function();
foo.bind === Function.prototype.bind
```

You can assign the bound function to be a (static) method in an object or as a prototype method; the value of `this` would remain the value passed in when the `bind` method was called, as seen in the following example:

```
// As a (static) object method
const bar = {
  foo: foo,
  boundFoo: boundFoo
};

bar.foo(); // bar
bar.boundFoo(); // 42

// As a prototype method
const Baz = function () {};
Baz.prototype.foo = foo;
Baz.prototype.boundFoo = boundFoo;
```

```
const myBaz = new Baz();
myBaz.foo(); // myBaz
myBaz.boundFoo(); // 42
```

Once you create a bound function, you cannot bind it again, as seen here:

```
const func = function () { console.log(this) };
const funcBoundTo42 = func.bind(42);
const funcBoundToBar = funcBoundTo42.bind("bar");

funcBoundTo42(); // 42
funcBoundToBar(); // 42 <- binding the bound function again has no
effect
```

call and apply

While the `bind` method returns a new function bound to a given context, the `call` and `apply` methods immediately invoke the function while passing in a context, as well as any arguments the function accepts. The only difference between the two methods is that `call` expects the arguments to be listed out directly, whereas `apply` expects the arguments as one array, as shown in the following snippet:

```
const foo = function (x, y) {
  console.log(this);
  console.log(x);
  console.log(y);
}

foo(); // window; undefined; undefined;
foo(1, 2); // window; 1; 2;
foo.call(0); // 0; undefined; undefined;
foo.call(0, 1); // 0; 1; undefined;
foo.call(0, 1, 2); // 0; 1; 2;
foo.apply(0, [1, 2]); // 0; 1; 2;
```

Arrow functions

Often, developers need to write `func.bind(this)` when calling a function. ECMAScript 2015 introduced a new syntax for passing in the current context to the function you're calling, which looks cleaner and less bloated than using the `bind` method every time, as follows:

```
function Foo () {
  window.setTimeout(function () {
```

```
      console.log(this); // window
    }, 0);

    window.setTimeout(function() {
      console.log(this); // Foo
    }.bind(this), 0);
    // Notice the () => {} arrow function here
    window.setTimeout(() => {
      console.log(this); // Foo
    }, 2000)
  }

  new Foo();
```

Context vs execution context

Note that *context* is different from **execution context**. The latter should be thought of in relation to the **execution stack**, which is how the JavaScript engine knows which part of the code it should execute next, and how it determines which variables are in **scope**.

When you first execute a script or program, it will initially be in the **global execution context**. When a function is invoked, it will cause a new execution context to be created and added to the top of the execution stack. If you execute another function *within* that function, then the inner function will also have an execution context, which sits on top of the outer function's execution context. The JavaScript engine will always execute the current execution context, which sits at the top of the execution stack. When a function completes, that function's execution context is popped off the top of the execution stack and the code in the execution stack below continues to execute until *it* is complete, and so on.

A function's execution context is associated with its corresponding **activation object** (**AO**) / **variable object** (**VO**), which is an object that contains the function's arguments as well as any variables and function declarations made inside the function body. When the function body is executed, identifiers are resolved by checking the AO associated with the function's execution context, as well as each execution context below the current one. This explains why the body of an inner function can access variables declared in the outer function.

Finally, although *context* and *execution context* are two different concepts, they *are* related. When a function is invoked, the value of `this` (and thus the context) is determined when the execution context is created, but before the function body is executed.

Summary

We have covered a lot of complicated subjects in this chapter, so let's do a quick recap to solidify your understanding.

First, we looked at data types in JavaScript. There are six primitive types and one object type. However, some primitive types have wrapper objects, which can make them behave like an object through a process called autoboxing. Functions are a special type of object and any function can act as a constructor function if called with the `new` keyword. Constructor functions encapsulate the logic required to create multiple objects with similar properties/methods.

Next, we looked at how inheritance works. In JavaScript, inheritance is prototype-based, not class-based. Every object (including functions) has a hidden `[[Prototype]]` property, which is exposed by browsers through the `__proto__` property; furthermore, every function also has a `prototype` property. When a new object is constructed using a constructor function, a reference to the function's `prototype` property is saved in the object's `__proto__` property. When looking for a method or property, the JavaScript engine will look through the prototype inheritance chain. In ES6, classes were added which desugar into constructor functions.

Lastly, we discussed how context works in JavaScript. We discussed how the `this` keyword represents the current context. When inside a function, the context changes depending on how the function is called, but you can bind the context within a function using `bind`, `call`, or `apply`.

This chapter gave you a comprehensive primer on the most fundamental (and most overlooked) principles of JavaScript. In the next chapter, we will get familiar with some new syntax introduced in the ECMAScript 2015 standard.

Index

P

S

T

V

W