

Constructing JavaScript Objects

Lesson Objectives

When you complete this lesson, you will be able to:

- *construct objects with your own constructors, and **new**.*
- *construct object literals.*
- *construct empty objects and add new properties.*
- *compare how a function works as a function, and as a constructor.*
- *initialize an object's property values in a constructor.*
- *use the conditional operator.*
- *explore the value of **this** when an object is created.*
- *construct arrays in two ways.*

Just about everything in JavaScript is an object, so understanding objects is key to understanding and programming JavaScript. In this lesson, we'll delve into how we create objects in JavaScript.

Constructing JavaScript Objects

When you construct an object in JavaScript, you are creating a dynamic collection of property names and values. You've already seen a couple of different ways to create objects, using a constructor function (like the `Book()` function we used in the previous lesson), and using object literals.

Let's take a closer look at three ways you can construct objects and how they are similar to and different from each other.


Constructing an Object with a Constructor Function

The first way to construct objects that we'll check out uses the a constructor function. In the previous lesson, we used a constructor function, `Book()`, to create book objects, passing in arguments for title, author, the date the book was published, and whether it had been made into a movie. We'll use that same object here, except we'll add a new method, `display()`, to the object:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Constructing objects </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
      this.title = title;
      this.author = author;
      this.published = published;
      this.hasMovie = hasMovie;
      this.display = function() {
        console.log(this);
      };
    }
    var book1 = new Book("The Hound of the Baskervilles", "Sir Arthur
Conan Doyle", 1901, true);
    book1.display();
  </script>
</head>
<body>
</body>
</html>
```

Save this in your in your work folder as *objectConstr.html*, and load it into a browser. Open the console (and reload the page if you need to), and you see that the constructor function created a book object:



```
▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle",
published: 1901, hasMovie: true, display: function} ⓘ
  ▶ display: function () {
    hasMovie: true
    published: 1901
    title: "The Hound of the Baskervilles"
  }
  ▶ __proto__: Book
```

(This screenshot is in the Chrome console, but it should look similar in IE, Firefox, and Safari).

A constructor function is just like any other function, but we *call* a constructor function differently from other functions: we use the word `new`.

OBSERVE:

```
var book1 = new Book("The Hound of the Baskervilles", "Sir Arthur Conan Doyle", 1901, true);
```

The word `new` makes all the difference. The `new` keyword indicates that we are using a function to construct an object, rather than just execute code (although a constructor function can do that too). Within the constructor function, we refer to the object that is being created as `this`:

OBSERVE:

```
function Book(title, author, published, hasMovie) {
  this.title = title;
  this.author = author;
  this.published = published;
  this.hasMovie = hasMovie;

  this.display = function() {
    console.log(this);
  };
}
```

When you call a function with `new`, inside that function, a new, empty object is created and the `this` keyword is set to that object. (Inside the function, `this` acts just like a local variable, except that you can't set its value yourself. That's done for you automatically). Then you use `this` to set the values of any properties you want in that object. At the end of the function, you don't have to explicitly return the object you're creating. JavaScript does that for you automatically because you used the `new` keyword when you called the function. The object that is returned is `this`, the new object that was created when you called the function and contains the properties you set in the function.

Of course any function *could* return an object if you wanted it to:

INTERACTIVE SESSION:

```
| > function makeObj() { return { x: 1 }; }
undefined
| > var myObj = makeObj();
undefined
| > myObj
Object {x: 1}
```

However, in this example, `makeObj()` is *not* a constructor function because we didn't call it with `new`. Instead we called the function normally, and inside the function, created an object literal on the fly, and returned it to the caller of the function `makeObj()`. These two ways of creating functions might seem similar, but there are a few key differences. The value of `this` inside `makeObj()` is *not* set to the object that's being created, and if you don't explicitly return an object from `makeObj()`, the default return value is `undefined`.

These differences in how functions behave, depending on whether you call them with `new` or not, is one reason why we always (as a convention) use an uppercase letter to begin the name of a constructor function (like `Book()`), but use a lowercase letter to begin the name of a regular function (like `makeObj()`). That way, you can tell at a glance at your code whether a function is designed to be a constructor function.

There's one other thing that happens when you create an object by calling a constructor function with `new` that doesn't happen when you create objects in other ways. Look back at the object we created by calling `makeObj()`:

```
> function makeObj() { return { x: 1 }; }
undefined
> var myObj = makeObj();
undefined
> myObj
Object {x: 1}
```

Now compare that to what you saw in the console earlier for the `book1` object (if you still have the page loaded, you can just type `book1` in the console to see it again, but make sure you do this in either the Chrome or Safari console specifically):

```
▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle",
  published: 1901, hasMovie: true, display: function} ⓘ
  ▶ display: function () {
    hasMovie: true
    published: 1901
    title: "The Hound of the Baskervilles"
  }
  ▶ __proto__: Book
```

When you display `myObj` in the console, you see the word `Object` next to the object:

OBSERVE:

```
Object {x: 1}
```

But you see the word "Book" next to the `book1` object:

OBSERVE:

```
Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle", published: 1901, hasMovie: true, display: function}
```

In the `Book` example, we created the `book1` object with the `Book()` constructor function. When you create an object with a constructor function, JavaScript keeps track of that function in a property called `constructor`. `constructor` is a property of the object, in this case `book1`, that results from calling the constructor function, `Book()`. Now, JavaScript also sets the `constructor` property for objects *not* created with a constructor function, like the literal object we created and returned from the `makeObj()` function, but in this case, the `constructor` property is set to `Object()`.

You can access the constructor for an object:

INTERACTIVE SESSION:

```
| > myObj.constructor  
function Object() { [native code] }  
| > book1.constructor  
function Book(title, author, published, hasMovie) {  
  this.title = title;  
  this.author = author;  
  this.published = published;  
  this.hasMovie = hasMovie;  
  
  this.display = function() {  
    console.log(this);  
  };  
}
```

You can think of the constructor function of an object, whether it's `Book()` or `Object()`, as determining the *type* of the object. This isn't strictly true like it is in statically typed languages like Java, but it can be a handy way to describe objects. We'll return to this idea in a later lesson when we talk about the `instanceof` operator.

For now, the key takeaway for you is to understand how constructing an object using a constructor function with the `new` keyword is different from other ways that we create objects.

Constructing an Object Using a Literal

You just saw an example of creating a literal object and returning it from a function. Let's create another literal object, another book, so we can compare the result directly with the `book1` object we created using a constructor function. Modify `objectConstr.html` as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Constructing objects </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
      this.title = title;
      this.author = author;
      this.published = published;
      this.hasMovie = hasMovie;

      this.display = function() {
        console.log(this);
      };
    }
    var book1 = new Book("The Hound of the Baskervilles", "Sir Arthur
Conan Doyle", 1901, true);
    book1.display();

    var book2 = {
      title: "The Adventures of Sherlock Holmes",
      author: "Sir Arthur Conan Doyle",
      published: 1892,
      movie: true,
      display: function() {
        console.log(this);
      }
    };
    book2.display();

  </script>
</head>
<body>
</body>
</html>
```

Save the changes to your file and load or refresh the page in your browser. Open the console, and compare `book1` and `book2` (using Chrome or Safari):

```
▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle", published: 1901, hasMovie: true, display: function} ⓘ  
  author: "Sir Arthur Conan Doyle"  
  ▶ display: function () {  
    hasMovie: true  
    published: 1901  
    title: "The Hound of the Baskervilles"  
  }  
  ▶ __proto__: Book  
▼ Object {title: "The Adventures of Sherlock Holmes", author: "Sir Arthur Conan Doyle", published: 1892, movie: true, display: function} ⓘ  
  author: "Sir Arthur Conan Doyle"  
  ▶ display: function () {  
    movie: true  
    published: 1892  
    title: "The Adventures of Sherlock Holmes"  
  }  
  ▶ __proto__: Object  
← undefined
```

These two objects are similar: both have the same property names, both have a `display()` method, and the types of all the property values are the same. However, if you look at the constructors for `book1` and `book2`, you'll see (just like in `myObj` earlier), that the constructor for `book1` is `Book()`, because we created it using a constructor function, but the constructor for `book2` is `Object()` because we created it using an object literal:

INTERACTIVE SESSION:

```
| > book1.constructor  
function Book(title, author, published, hasMovie) {  
  this.title = title;  
  this.author = author;  
  this.published = published;  
  this.hasMovie = hasMovie;  
  
  this.display = function() {  
    console.log(this);  
  };  
}  
| > book2.constructor  
function Object() { [native code] }
```

The `[native code]` means that the implementation of the `Object()` constructor is hidden because it's internal to the browser.

The property in the objects called `__proto__` is the last property in both the `book1` and `book2` objects:

```
▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle", published: 1901, hasMovie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    hasMovie: true
    published: 1901
    title: "The Hound of the Baskervilles"
  }
  ▶ __proto__: Book
▼ Object {title: "The Adventures of Sherlock Holmes", author: "Sir Arthur Conan Doyle", published: 1892, movie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    movie: true
    published: 1892
    title: "The Adventures of Sherlock Holmes"
  }
  ▶ __proto__: Object
< undefined
```

The value of `__proto__` for `book1` is `Book`, and the value of `__proto__` for `book2` is `Object`. You might be thinking that the `__proto__` property must be related to the constructor function for an object, and you'd be right (although they are *not* the same thing).

You also might notice that the `constructor` property is *not* listed in the `book1` and `book2` objects' properties. That's because this property is *inherited* from the object's `prototype`. (As you might guess, the `__proto__` property is also related to the prototype). We'll talk more about prototypes in a later lesson.

What about `this` in a literal object? You already know that you can use `this` in a method of an object to refer to "this object," but unlike in a constructor function, we don't (and can't) use `this` to initialize object properties. Instead, we create properties and initialize them by specifying the name/value pairs in an object, by literally typing them. (That's why it's called an object literal). The only time `this` refers to the object is when you call one of its methods. (How `this` gets set and what it gets set to is yet another topic we'll come back to in more depth later).

Constructing an Object Using a Generic Object Constructor

Another way to create an object literal is to start with an empty object, and then add properties to it. You can create an empty, generic object in one of two ways:

OBSERVE:

```
var obj1 = { };
var obj2 = new Object();
```

Both of these approaches to create an object do the same thing. You'll see an empty object created the first way more often (because it's a little easier to write), but it's instructive to understand the second way as well. When you create an object with `new Object()`, it's just

like when you create an object with `new Book()`, except that `Object()` is a built-in constructor function that you don't have to write yourself. You don't pass any arguments to `Object()` to initialize object properties in the constructor function, instead, you add them all after the object is created. Modify *objectConstr.html* as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Constructing objects </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
      this.title = title;
      this.author = author;
      this.published = published;
      this.hasMovie = hasMovie;

      this.display = function() {
        console.log(this);
      };
    }
    var book1 = new Book("The Hound of the Baskervilles", "Sir Arthur
Conan Doyle", 1901, true);
    book1.display();

    var book2 = {
      title: "The Adventures of Sherlock Holmes",
      author: "Sir Arthur Conan Doyle",
      published: 1892,
      movie: true,
      display: function() {
        console.log(this);
      }
    };
    book2.display();

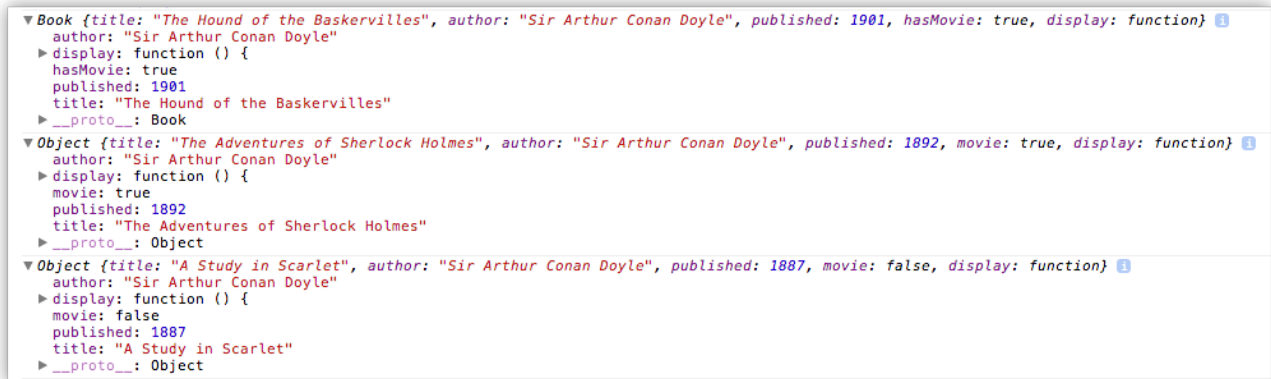
    var book3 = new Object(); // same as var book3 = { };
    book3.title = "A Study in Scarlet";
    book3.author = "Sir Arthur Conan Doyle";
    book3.published = 1887;
    book3.movie = false;
    book3.display = function() {
      console.log(this);
    };
    book3.display();

  </script>
</head>
```

```
<body>
</body>
</html>
```

We added the exact same properties that we added to `book1` and `book2`, only with some different values, because it's a different book. `book3` also has a `display()` method, just like `book1` and `book2`.

Save your changes, and load or refresh the page in your browser. In the console, compare `book3` with `book2` and `book1`.



```
▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle", published: 1901, hasMovie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    hasMovie: true
    published: 1901
    title: "The Hound of the Baskervilles"
  }
  ▶ __proto__: Book
▼ Object {title: "The Adventures of Sherlock Holmes", author: "Sir Arthur Conan Doyle", published: 1892, movie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    movie: true
    published: 1892
    title: "The Adventures of Sherlock Holmes"
  }
  ▶ __proto__: Object
▼ Object {title: "A Study in Scarlet", author: "Sir Arthur Conan Doyle", published: 1887, movie: false, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    movie: false
    published: 1887
    title: "A Study in Scarlet"
  }
  ▶ __proto__: Object
```

`book3` looks similar to `book2`, because `book3` is also an object literal; it was just created in a slightly different way. Notice that the constructor for `book3` is also `Object()`, which you can test in the browser console:

INTERACTIVE SESSION:

```
| > book3.constructor
function Object() { [native code] }
```

So, *What's the Best Way to Make an Object?*

You've seen three different ways to construct an object (you'll see a fourth in a later lesson), but which is the *best* way?

That depends on the situation. If all you need is a quick, one-off object, then creating an object literal like we did with `book2` or `book3` is probably good enough. However, if you know that you're going to need multiple book objects, writing a constructor function like `Book()`, that you can use to make many book objects is a better choice. You'll also want to consider whether the objects you're creating are, say, `Books` or `Magazines`. As you've seen, objects created with

a constructor function have that extra information about how they were created, which can be useful. We'll see an example of that in a later lesson, when we talk about prototypes.

Initializing Values in Constructors

Let's go back to constructor functions now and look at ways you can initialize the properties of the object you're constructing with the function. We'll use a different example, so open a new file:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Initializing objects </title>
  <meta charset="utf-8">
  <script>
    function Point(x, y) {
      if (x == undefined || x == null) {
        this.x = 50;
      } else {
        this.x = x;
      }

      if (y == undefined || y == null) {
        this.y = 50;
      } else {
        this.y = y;
      }

      // Make a toString() method we can use to display the point
      this.toString = function() {
        return "[" + this.x + ", " + this.y + "]";
      }
    }

    // Can have code in constructors too
    var p = new Point();
    console.log("My point is: " + p.toString());
  </script>
</head>
<body>
</body>
</html>
```

Save this new file in your work folder and name it *point.html*. Load *point.html* in your browser. the Point object, **p**, is displayed like this:

OBSERVE:

```
My point is: [50, 50]
```

In this code, we've got a `Point()` constructor function to create Point objects. Let's discuss the code:

OBSERVE:

```
function Point(x, y) {
  if (x == undefined || x == null) {
    this.x = 50;
  } else {
    this.x = x;
  }

  if (y == undefined || y == null) {
    this.y = 50;
  } else {
    this.y = y;
  }

  // toString(): display the point
  this.toString = function() {
    return "[" + this.x + ", " + this.y + "];"
  }
}

var p = new Point();
console.log("My point is: " + p.toString());
```

We use a constructor function, `Point()` to create a Point object, **p**. So, if you look at the constructor property of the object **p**, you'll see the `Point()` function.

`Point()` actually expects two arguments, **x** and **y**, which are the coordinates of the point, but we call `Point()` with no arguments. It turns out that JavaScript is totally okay with this, but unless we do something further, the **x** and **y** coordinates of the point we're trying to create will be undefined. So, we write code to test to see whether the **x** and **y** values are passed in. If they are, we initialize `this.x` and `this.y` with the values **x** and **y** respectively. If we don't pass in any values, we use the default value, 50 for both `this.x` and `this.y`.

Now, you might be tempted to take a shortcut and rewrite `if (x == undefined || x == null) { ... }` as `if (!x) { ... }` (based on what you learned in the previous lesson about truthy and falsey values), but be careful! We might want a point at 0, 0, and 0 is falsey, so that shortcut won't work for us in this case. We can, however, shorten the initialization a bit by making use of the *conditional operator*. Modify `point.html` as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Initializing objects </title>
  <meta charset="utf-8">
  <script>
    function Point(x, y) {
      this.x = (!x && x != 0) ? 50 : x;
      this.y = (!y && y != 0) ? 50 : y;

      if (x == undefined || x == null) {
        this.x = 50;
      } else {
        this.x = x;
      }

      if (y == undefined || y == null) {
        this.y = 50;
      } else {
        this.y = y;
      }

      // Make a toString() method we can use to display the point
      this.toString = function() {
        return "[" + this.x + ", " + this.y + "]";
      }
    }

    // Can have code in constructors too
    var p = new Point();
    console.log("My point is: " + p.toString());
  </script>
</head>
<body>
</body>
</html>
```

Save these changes, and refresh or load `point.html` in your browser. You see the same result as before.

Now, some programmers avoid the conditional operator (also written `? :` for short) like the plague, because it's harder to read, and you can always write the same code using an easier-to-read `if/else` statement. If you're in this camp, feel free to use `if/else` statements instead. Still, you need to know how to read statements that use the conditional operator too. They are used fairly often to initialize objects.

So, you read this:

OBSERVE:

```
this.x = (!x && x != 0) ? 50 : x;
```

like this: "If not `x` AND `x` is not equal to 0, THEN set `this.x` to 50 ELSE set `this.x` to `x`."

In other words, you read `?` as THEN, and `:` as ELSE.

This statement checks to see if `!x` is true, which it will be if the parameter `x` is null, undefined, or 0. Then, to handle the 0 case, we check to make sure `x != 0`. If `x` is 0, this returns false, so the whole conditional is false, and we set `this.x` equal to `x`, which is 0 in this case. If `x` is undefined or null, we set `this.x` to 50. If `x` is a non-zero number we set `this.x` to `x`.

Don't get the *parameter* `x` mixed up with the *property* `this.x`. They are two different variables! Remember that we're passing a value into the constructor to initialize the property `this.x`. The value we pass in gets assigned to the parameter `x`.

Each `Point` object also has a method, `toString()`, that creates a string representation of the point for display. In `toString()` we use `this.x` and `this.y` to create the string representing the `Point`. Make sure to use the object's properties, and *not* the parameters in the method. If you forget, and use `x` and `y` instead of `this.x` and `this.y`, what could happen? Well, if `Point()` is called with no arguments, then the parameters `x` and `y` will be undefined. While the `Point`'s `x` and `y` properties are set correctly to 50 each, you'll see `[undefined, undefined]` when you call the `toString()` method.

this

We've talked a bit about what happens to `this` when you're constructing objects. To make sure you've got a handle on `this` when you're working with constructor functions, regular functions, objects, and methods, let's take a look at another example. Create this new file:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> What happens to this </title>
  <meta charset="utf-8">
  <script>
    //
    // Rectangle constructor that makes rectangle objects
    //
    function Rectangle(width, height) {
      console.log("This in Rectangle is: ");
      console.log(this);

      this.width = width || 0;
      this.height = height || 0;
      this.getArea = function() {
        console.log("This in Rectangle's getArea is: ");
        console.log(this);
        return this.width * this.height;
      };
    }

    var rect1 = new Rectangle(5, 10);
    console.log("Area of rectangle 1: " + rect1.getArea());

    //
    // A function that makes rectangle objects
    //
    function makeRectangle(width, height) {
      console.log("This in makeRectangle is: ");
      console.log(this);

      return {
        width: width || 0,
        height: height || 0,
        getArea: function() {
          console.log("This in makeRectangle's getArea is: ");
          console.log(this);
          return this.width * this.height;
        }
      };
    }

    var rect2 = makeRectangle(5, 10);
    console.log("Area of rectangle 2: " + rect2.getArea());

    // getArea function
    function getArea(r) {
      console.log("This in getArea is: ");
```

```

        console.log(this);
        return (r.width * r.height);
    }
    console.log("Area from getArea(rect1): " + getArea(rect1));
</script>
</head>
<body>
</body>
</html>

```

Save this file in your work folder as *rectangle.html* and load it into your browser. In the console, you see lots of output:

```

This in Rectangle is:
Rectangle {}
This in a Rectangle's getArea is:
▶ Rectangle {width: 5, height: 10, getArea: function}
Area of rectangle 1: 50
This in makeRectangle is:
▶ Window {top: Window, window: Window, location: Location, external: Object, chrome: Object...}
This in a rectangle's getArea is:
▶ Object {width: 5, height: 10, getArea: function}
Area of rectangle 2: 50
This in getArea is:
▶ Window {top: Window, window: Window, location: Location, external: Object, chrome: Object...}
Area from getArea(rect1): 50

```

There's quite a bit going on here, so we'll step through it one piece at a time:

OBSERVE:

```

//
// Rectangle constructor that makes rectangle objects
//
function Rectangle(width, height) {
    console.log("This in Rectangle is: ");
    console.log(this);

    this.width = width || 0;
    this.height = height || 0;
    this.getArea = function() {
        console.log("This in a Rectangle's getArea is: ");
        console.log(this);
        return this.width * this.height;
    };
}
var rect1 = new Rectangle(5, 10);

```



```
console.log("Area of rectangle 1: " + rect1.getArea());
```

In this code, we've got a `Rectangle()` constructor function that we can use to make rectangles with `width` and `height` properties, and a method, `getArea()` that returns the area of the rectangle. We've added calls to `console.log()` in two different places within the constructor function to inspect the value of `this`.

When we call `new Rectangle(5, 10)` to create a rectangle object, `rect1`, the first two lines of code in the function display the value of `this`. The result is an empty `Rectangle` object:

OBSERVE:

```
This in Rectangle is:  
Rectangle {}
```

When we call a construction function with `new`, the first thing that happens is a new, empty object is created. This is the `Rectangle {}` object we see here in the console. Its constructor is `Rectangle`, and it doesn't have any properties (yet).

The rest of the constructor function assigns values to the `width`, `height` and `getArea()` properties, so that when the object is returned at the end of the function, all of its properties have been created and given values.

Next, we call the `getArea()` method of the rectangle object we just created. The first two lines of the `getArea()` method display the value of `this` in the console. We see that `this` is a `Rectangle` object, and that now it has the properties we created in the constructor:

OBSERVE:

```
This in Rectangle is:  
Rectangle {}  
This in Rectangle's getArea is:  
Rectangle {width: 5, height: 10, getArea: function}  
Area of rectangle 1: 50
```

Finally, we display the result of the call to `getArea()`, which is 50.

So in both the body of the constructor function and the method, `this` refers to "this object," that is, the `Rectangle` object created by the constructor. The first use of `this` is the object when it's created and modified at object creation time (when we call the constructor function). The second use of `this` is the object when it's accessed after we call the object's method,

`getArea()`. In this case, the value of `this` is assigned automatically because you are calling a method of an object.

Now let's compare that to what happens when we call `makeRectangle()` to make a rectangle object.

OBSERVE:

```
//  
// A function that makes rectangle objects  
//  
function makeRectangle(width, height) {  
  console.log("This in makeRectangle is: ");  
  console.log(this);  
  
  return {  
    width: width || 0,  
    height: height || 0,  
    getArea: function() {  
      console.log("This in makeRectangle's getArea is: ");  
      console.log(this);  
      return this.width * this.height;  
    }  
  };  
}  
  
var rect2 = makeRectangle(5, 10);  
console.log("Area of rectangle 2: " + rect2.getArea());
```

`makeRectangle()` isn't a constructor function, it's just a regular function, so we don't call it with `new`; we just call it the regular way. The first two lines of code in `makeRectangle()` display the value of `this`. You can see that `this` is the global, `Window` object:

OBSERVE:

```
This in makeRectangle is:  
Window {top: Window, window: Window, location: Location, external:  
Object, chrome: Object}  
This in a makeRectangle's getArea is:  
Object {width: 5, height: 10, getArea: function}  
Area of rectangle 2: 50
```

There is no object being created automatically by `makeRectangle()`. While we are creating an object in this function, that object is not the value of `this`. We create that object in the next statement (by returning an object literal).

However, when we call the `getArea()` method of the object returned by `makeObject()`, `rect2`, you can see that the value of `this` in the `getArea()` method is indeed an object:

OBSERVE:

```
This in makeRectangle is:  
Window {top: Window, window: Window, location: Location, external:  
Object, chrome: Object}  
This in makeRectangle's getArea is:  
Object {width: 5, height: 10, getArea: function}  
Area of rectangle 2: 50
```

The object that we see is `rect2`, "this object," that is, the object whose method we called. Again, notice that the object's constructor is `Object()` (compare to the constructor for `rect1` above). Finally, we display the area for `rect2`, which is 50.

We've also included a function `getArea()` that takes a rectangle object and returns the area:

OBSERVE:

```
// getArea function  
function getArea(r) {  
    console.log("This in getArea is: ");  
    console.log(this);  
    return (r.width * r.height);  
}  
console.log("Area from getArea(rect1): " + getArea(rect1));
```

The value of `this` in the `getArea()` function displays as:

OBSERVE:

```
This in getArea is:  
Window {top: Window, window: Window, location: Location, external:  
Object, chrome: Object}  
Area from getArea(rect1): 50
```

Once again, this is the global `Window` object. Just like `makeRectangle()`, `getArea()` is just a regular old function that happens to take an object and compute something with it. There is no "this object" for this function, so the value of `this` gets set to the `Window` object automatically.

Keeping track of the value of `this` can be tricky in JavaScript, but it's important. You'll need to understand how the value of `this` is set, and what it's set to in all situations. We'll also revisit `this` in later lessons.

Constructing Array Objects

Before we leave this lesson, let's talk about constructing Array objects. Arrays are objects, although you should think of them as a special kind of object with features that the objects we've been creating so far don't have, like an index, and ordering imposed on the items in the object.

There are two ways to create an Array object. Type these commands in the console:

INTERACTIVE SESSION:

```
| > var a1 = new Array();  
undefined  
| > a1[0] = 1;  
1  
| > a1[1] = 2;  
2  
| > a1[2] = 3;  
3  
| > a1  
[1, 2, 3]
```

Here we created an empty array, `a1`, using the `Array()` constructor function, calling it with `new` like we would any other constructor function. Then we add array items one at a time to the 0, 1, and 2 indices in the array. This is analogous to using `new Object()` and adding object properties one at a time, like we did with `book3` earlier in the lesson.

You can use bracket notation to access an object's properties, like this:

OBSERVE:

```
var theTitle = book1["title"];
```

The bracket notation is used to access the items in an array, except we use an index instead of a property name. The second way to create an Array is to use the array literal notation:

INTERACTIVE SESSION:

```
| > var a2 = [1, 2, 3];  
undefined  
| > a2  
[1, 2, 3]
```

This does exactly the same thing as the previous example. It creates a new array with values at the 0, 1, and 2 indices, but it's a lot shorter to write! In practice, you'll rarely use the `Array()` constructor to create an array. Instead you'll use the more concise array literal notation. One exception is when you need to create an empty array with a predefined number of indices:

INTERACTIVE SESSION:

```
| > var a3 = new Array(100);  
undefined  
| > a3  
[undefined x 100]
```

This creates an array with length 100, with all the items at every index set to `undefined`, and the Chrome console uses the shorthand "[undefined x 100]" to display the value of this array. (Other browsers don't use this shorthand, so you'll see different results in different browsers when you ask for the value of `a3`).

Just like any other object, arrays can have named properties, and in fact, come with a named property, `length`, that you'll use to get the length of your array:

INTERACTIVE SESSION:

```
| > a1.length  
3  
| > a2.length  
3  
| > a3.length  
100
```

Just like other objects, you can use the `constructor` property to inspect the constructor function for the array:

INTERACTIVE SESSION:

```
| > a1.constructor  
function Array() { [native code] }  
| > a2.constructor  
function Array() { [native code] }  
| > a3.constructor  
function Array() { [native code] }
```

In each case, the constructor for the array is `Array()`. This is analogous to `Object()` being the constructor for objects created with literal notation or with `new Object()`.

In this lesson, you learned about constructing objects, how constructor functions work, the difference between objects created with a constructor function and those created using literal notation, and what happens to `this` when you construct and use an object. In the next lesson, we'll explore more object-related goodies: prototypes and inheritance.

More about the material in this lesson

The Riverside JS Workshop would like to acknowledge the generosity of O'Reilly Media, Inc. for making this material available to us through the Creative Commons License. We would also like to acknowledge the great work of Elisabeth Robson who authored this content. She is one of our favorites. Elisabeth has written a number of *Head First* programming books for web developers. We recommend them highly. You can browse her work [here](#).



Copyright © 1998-2015 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

The original source document has been altered. It has been edited to accommodate this format and enhance readability.