Understanding Classes in JavaScript

This article was originally written for DigitalOcean.

Introduction

In Understanding Prototypes and Inheritance in JavaScript, we learned how JavaScript is a prototype-based language, and every object in JavaScript has a hidden internal property called `[[Prototype]]` that can be used to extend object properties and methods.

Until recently, industrious developers used constructor functions to mimic an object-oriented design pattern in JavaScript. The language specification ECMAScript 2015, often referred to as ES6, introduced classes to the JavaScript language. Classes are often described as "syntactic sugar" over prototypes and inheritance, which means they offer a cleaner and easier syntax without offering new functionality.

Classes Are Functions

A JavaScript class is a type of function. Classes are declared with the `class` keyword. We will use function expression syntax to initialize a function and class expression syntax to initialize a class.

```
// Initializing a function with a function expression
const x = function () {}

// Initializing a class with a class expression
const y = class {}
```

Previously, we learned that we can access the `[[Prototype]]` of an object using the `Object.getPrototypeOf()` method. Now let's test that out on the empty function we created.

```
Object.getPrototypeOf(x)
```

```
ƒ () { [native code] }
```

```
Object.getPrototypeOf(y)
```

```
ƒ () { [native code] }
```

We can see that the code declared with `function` and `class` both return a function `[[Prototype]]`. With prototypes, we learned that any function can become a constructor instance using the `new` keyword.

```
const x = function () {}

// Initialize a constructor from a function
const constructorFromFunction = new x()

console.log(constructorFromFunction)
```

```
x {}
constructor: ƒ ()
```

This applies to classes as well.

```
const y = class {}

// Initialize a constructor from a class
const constructorFromClass = new y()

console.log(constructorFromClass)
```

```
y {}
constructor: class
```

These prototype constructor examples are otherwise empty, but we can see how underneath the syntax, both methods are achieving the same end result.

In the prototypes and inheritance tutorial, we created an example based around character creation in a text-based role-playing game. Let's continue with that example here to update the syntax from functions to classes.

Originally, a constructor function would be initialized with a number of parameters, which would be assigned as properties of `this`, referring to the function itself. The first letter of the identifier would be capitalized by convention.

```
// Initializing a constructor function
function Hero(name, level) {
  this.name = name
  this.level = level
}
```

The new class syntax, shown below, is structured very similarly.

class.js

```
// Initializing a class definition
class Hero {
  constructor(name, level) {
    this.name = name
    this.level = level
  }
}
```

We know a constructor function is meant to be an object blueprint by the capitalization of the first letter of the initializer (which is optional) and through familiarity with the syntax. The `class` keyword communicates in a more straightforward fashion the objective of our function.

The only difference in the syntax of the initialization is using the `class` keyword instead of `function`, and assigning the properties inside a `constructor()` method.

Defining Methods

The common practice with constructor functions is to assign methods directly to the `prototype`, instead of in the initialization, as seen in the `greet()` method below.

constructor.js

```
}

// Adding a method to the constructor
Hero.prototype.greet = function () {
  return `${this.name} says hello.`
}
```

With classes this syntax is simplified, and the method can be added directly to the class. Using the method definiton shorthand introduced in ES6, defining a method is an even more concise process.

class.js

```
class Hero {
  constructor(name, level) {
    this.name = name
    this.level = level
  }

  // Adding a method to the constructor
  greet() {
    return `${this.name} says hello.`
  }
}
```

Let's take a look at these properties and methods in action. We will create a new instance of `Hero` using the `new` keyword, and assign some values.

```
const hero1 = new Hero('Varg', 1)
```

If we print out more information about our new object with `console.log(hero1)`, we can see more details about what is happening with the class initialization.

```
Hero {name: "Varg", level: 1}
__proto__:
  ▶ constructor: class Hero
  ▶ greet: ƒ greet()
```

We can see in the output that the `constructor()` and `greet()` functions were applied to the `__proto__`, or `[[Prototype]]` of `hero1`, and not directly as a method on the `hero1` object. While this

Extending a Class

An advantageous feature of constructor functions and classes is that they can be extended into new object blueprints based off the parent. This prevents repetition of code for objects that are similar but need some additional or more specific features.

New constructor functions can be created from the parent using the call method. In the example below, we will create a more specific character class called `Mage`, and assign the properties of `Hero` to it using `call()`, as well as adding an additional property.

We will also set the `Mage`'s prototype to the one of `Hero`. Thus, `Mage` will inherit the methods of the `Hero`'s prototype.

constructor.js

```
// Creating a new constructor from the parent
function Mage(name, level, spell) {
  // Chain constructor with call
  Hero.call(this, name, level)

  this.spell = spell
}

// Creating a new object using Hero's prototype as the prototype for the newly created c
Mage.prototype = Object.create(Hero.prototype)
```

Now we can create a new instance of `Mage` using the same properties as `Hero` as well as a new one we added.

```
const hero2 = new Mage('Lejon', 2, 'Magic Missile')
```

Sending `hero2` to the console, we can see we have created a new `Mage` based off the constructor.

```
Mage {name: "Lejon", level: 2, spell: "Magic Missile"}
__proto__:
```

With ES6 classes, the `super` keyword is used in place of `call` to access the parent functions. We will use `extends` to refer to the parent class.

class.js

```
// Creating a new class from the parent
class Mage extends Hero {
  constructor(name, level, spell) {
    // Chain constructor with super
    super(name, level)

    // Add a new property
    this.spell = spell
  }
}
```

Now we can create a new `Mage` instance in the same manner.

```
const hero2 = new Mage('Lejon', 2, 'Magic Missile')
```

We will print `hero2` to the console and view the output.

```
Mage {name: "Lejon", level: 2, spell: "Magic Missile"}
__proto__: Hero
    ▶ constructor: class Mage
```

The output is nearly exact the same, except in the case of classes the `[[Prototype]]` is linked to the parent, in this case `Hero`.

Below is a side-by-side comparison of the entire process of initialization, adding methods, and inheritance of a constructor function and a class.

constructor.js

```
function Hero(name, level) {
  this.name = name
  this.level = level
}

// Adding a method to the constructor
```

```
// Creating a new constructor from the parent
function Mage(name, level, spell) {
  // Chain constructor with call
  Hero.call(this, name, level)

  this.spell = spell
}

// Creating a new object using Hero's prototype as the prototype for the newly created c
Mage.prototype = Object.create(Hero.prototype)
```

class.js

```
// Initializing a class
class Hero {
  constructor(name, level) {
    this.name = name
    this.level = level
  }

  // Adding a method to the constructor
  greet() {
    return `${this.name} says hello.`
  }
}

// Creating a new class from the parent
class Mage extends Hero {
  constructor(name, level, spell) {
    // Chain constructor with super
    super(name, level)

    // Add a new property
    this.spell = spell
  }
}
```

Although the syntax is quite different, the underlying result is nearly the same between both methods. Classes give us an easier, more concise way of creating object blueprints, and constructor functions describe more accurately what is happening under the hood.

In this article, we learned about the similarities and differences between JavaScript constructor functions and ES6 classes. Both classes and constructors imitate an object-oriented inheritance model to JavaScript, which is a prototype-based inheritance language.

Understanding prototypical inheritance is paramount to being an effective JavaScript developer. Being familiar with classes is extremely helpful, as popular JavaScript libraries such as React make frequent use of the `class` syntax.

To learn more about classes, view the Class Reference on the Mozilla Developer Network.

Comments

4 Comments - *powered by utteranc.es*

vivianequeiroz commented on Feb 12, 2021

Such a nice explanation and great examples!
Thanks a lot for sharing your knowlogde, Tania :)

👍 1

anastasiosPou commented on Apr 15, 2021

I have a question about the Mage's prototype object creation.
`Mage.prototype = Object.create(Hero.prototype)`
If I try to make an instance of Mage, the instance's `__proto__.constructor` property points, in my opinion correctly, to the Hero constructor. When you console the `hero2` instance it says Mage.
How's that possible? I had to set `Mage.prototype.constructor = Mage` .

👍 1

TifeLaflame commented on May 7, 2021

Thank you for this article Tania, I started from "Build a Simple MVC App From Scratch in JavaScript" and I found myself just reading more articles.

SantosPaulo commented on Jul 8, 2021

Very nice article. Thank you, and good work

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub

ABOUT ME

Hello and thanks for visiting! My name is Tania Rascia, and this is my website and digital garden.

I'm a software developer who writes articles and tutorials about things that interest me. This site is and has always been free of ads, trackers, social media, affiliates, and sponsored posts.

I hope you enjoy the post and have a nice day.

DETAILS

- Published May 04, 2018

CATEGORY

- JavaScript

TAGS

javascript fundamentals