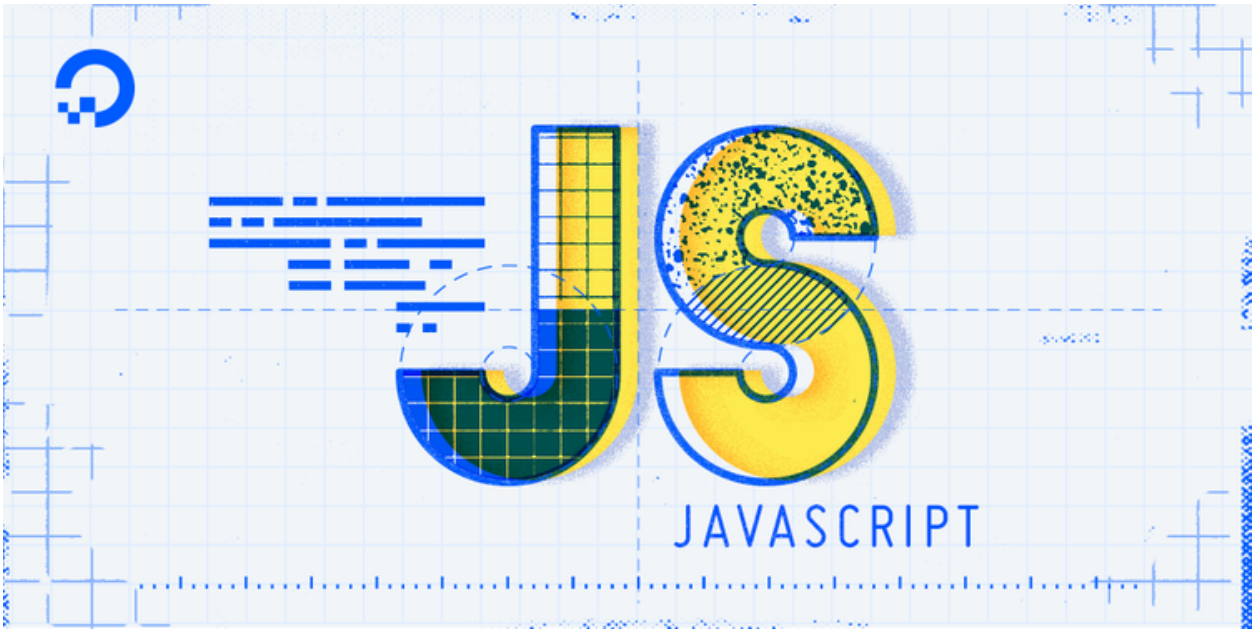

Understanding Prototypes and Inheritance in JavaScript

Posted January 12, 2018 80.2k JAVASCRIPT DEVELOPMENT

By: Tania Rascia

Introduction

JavaScript is a **prototype-based language**, meaning object properties and methods can be shared through generalized objects that have the ability to be cloned and extended. This is known as prototypical inheritance and differs from class inheritance. Among popular object-oriented programming languages, JavaScript is relatively unique, as other prominent languages such as PHP, Python, and Java are class-based languages, which instead define classes as blueprints for objects.

In this tutorial, we will learn what object prototypes are and how to use the constructor function to extend prototypes into new objects. We will also learn about inheritance and the prototype chain.

JavaScript Prototypes

In Understanding Objects in JavaScript, we went over the object data type, how to create an object, and how to access and modify object properties. Now we will learn how prototypes can be used to extend objects.

Every object in JavaScript has an internal property called `[[Prototype]]`. We can demonstrate this by creating a new, empty object.

```
let x = {};
```

This is the way we would normally create an object, but note that another way to accomplish this is with the object constructor: `let x = new Object()`.

> The double square brackets that enclose `[[Prototype]]` signify that it is an internal property, and cannot be accessed directly in code.

To find the `[[Prototype]]` of this newly created object, we will use the `getPrototypeOf()` method.

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

The output will consist of several built-in properties and methods.

```
Output
{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, …}
```

Another way to find the `[[Prototype]]` is through the `__proto__` property. `__proto__` is a property that exposes the internal `[[Prototype]]` of an object.

> It is important to note that `.__proto__` is a legacy feature and should not be used in production code, and it is not present in every modern browser. However, we can use it throughout this article for demonstrative purposes.

```
x.__proto__;
```

The output will be the same as if you had used `getPrototypeOf()`.

```
Output
{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, …}
```

It is important that every object in JavaScript has a `[[Prototype]]` as it creates a way for any two or more objects to be linked.

Objects that you create have a `[[Prototype]]`, as do built-in objects, such as `Date` and `Array`. A reference can be made to this internal property from one object to another via the `prototype` property, as we will see later in this tutorial.

Prototype Inheritance

When you attempt to access a property or method of an object, JavaScript will first search on the object itself, and if it is not found, it will search the object's `[[Prototype]]`. If after consulting both the object and its `[[Prototype]]` still

no match is found, JavaScript will check the prototype of the linked object, and continue searching until the end of the prototype chain is reached.

At the end of the prototype chain is `Object.prototype`. All objects inherit the properties and methods of `Object`. Any attempt to search beyond the end of the chain results in `null`.

In our example, `x` is an empty object that inherits from `Object`. `x` can use any property or method that `Object` has, such as `toString()`.

```
x.toString();
```

```
Output
[object Object]
```

This prototype chain is only one link long. `x` -> `Object`. We know this, because if we try to chain two `[[Prototype]]` properties together, it will be `null`.

```
x.__proto__.__proto__;
```

```
Output
null
```

Let's look at another type of object. If you have experience Working with Arrays in JavaScript, you know they have many built-in methods, such as `pop()` and `push()`. The reason you have access to these methods when you create a new array is because any array you create has access to the properties and methods on the `Array.prototype`.

We can test this by creating a new array.

```
let y = [];
```

Keep in mind that we could also write it as an array constructor, `let y = new Array()`.

If we take a look at the `[[Prototype]]` of the new `y` array, we will see that it has more properties and methods than the `x` object. It has inherited everything from `Array.prototype`.

```
y.__proto__;
```

```
[constructor: ƒ, concat: ƒ, pop: ƒ, push: ƒ, …]
```

You will notice a `constructor` property on the prototype that is set to `Array()`. The `constructor` property returns the constructor function of an object, which is a mechanism used to construct objects from functions.

We can chain two prototypes together now, since our prototype chain is longer in this case. It looks like `y` -> `Array` -> `Object`.

```
y.__proto__.__proto__;
```

```
Output
{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, …}
```

This chain is now referring to `Object.prototype`. We can test the internal `[[Prototype]]` against the `prototype` property of the constructor function to see that they are referring to the same thing.

```
y.__proto__ === Array.prototype;            // true
y.__proto__.__proto__ === Object.prototype; // true
```

We can also use the `isPrototypeOf()` method to accomplish this.

```
Array.prototype.isPrototypeOf(y);      // true
Object.prototype.isPrototypeOf(Array); // true
```

We can use the `instanceof` operator to test whether the `prototype` property of a constructor appears anywhere within an object's prototype chain.

```
y instanceof Array; // true
```

To summarize, all JavaScript objects have a hidden, internal `[[Prototype]]` property (which may be exposed through `__proto__` in some browsers). Objects can be extended and will inherit the properties and methods on `[[Prototype]]` of their constructor.

These prototypes can be chained, and each additional object will inherit everything throughout the chain. The chain ends with the `Object.prototype`.

Constructor Functions

Constructor functions are functions that are used to construct new objects. The `new` operator is used to create new instances based off a constructor function. We have seen some built-in JavaScript constructors, such as `new Array()` and `new Date()`, but we can also create our own custom templates from which to build new objects.

As an example, let's say we are creating a very simple, text-based role-playing game. A user can select a character and then choose what character class they will have, such as warrior, healer, thief, and so on.

Since each character will share many characteristics, such as having a name, a level, and hit points, it makes sense to create a constructor as a template. However, since each character class may have vastly different abilities, we want to make sure

each character only has access to their own abilities. Let's take a look at how we can accomplish this with prototype inheritance and constructors.

To begin, a constructor function is just a regular function. It becomes a constructor when it is called on by an instance with the `new` keyword. In JavaScript, we capitalize the first letter of a constructor function by convention.

characterSelect.js

```
// Initialize a constructor function for a new Hero
function Hero(name, level) {
  this.name = name;
  this.level = level;
}
```

We have created a constructor function called `Hero` with two parameters: `name` and `level`. Since every character will have a name and a level, it makes sense for each new character to have these properties. The `this` keyword will refer to the new instance that is created, so setting `this.name` to the `name` parameter ensures the new object will have a `name` property set.

Now we can create a new instance with `new`.

```
let hero1 = new Hero('Bjorn', 1);
```

If we console out `hero1`, we will see a new object has been created with the new properties set as expected.

```
Output
Hero {name: "Bjorn", level: 1}
```

Now if we get the `[[Prototype]]` of `hero1`, we will be able to see the `constructor` as `Hero()`. (Remember, this has the same input as `hero1.__proto__`, but is the proper method to use.)

```
Object.getPrototypeOf(hero1);
```

```
Output
constructor: ƒ Hero(name, level)
```

You may notice that we've only defined properties and not methods in the constructor. It is a common practice in JavaScript to define methods on the prototype for increased efficiency and code readability.

We can add a method to `Hero` using `prototype`. We'll create a `greet()` method.

characterSelect.js

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

Since `greet()` is in the `prototype` of `Hero`, and `hero1` is an instance of `Hero`, the method is available to `hero1`.

```
hero1.greet();
```

```
Output
"Bjorn says hello."
```

If you inspect the `[[Prototype]]` of Hero, you will see `greet()` as an available option now.

This is good, but now we want to create character classes for the heroes to use. It wouldn't make sense to put all the abilities for every class into the `Hero` constructor, because different classes will have different abilities. We want to create new constructor functions, but we also want them to be connected to the original `Hero`.

We can use the `call()` method to copy over properties from one constructor into another constructor. Let's create a Warrior and a Healer constructor.

characterSelect.js

```
...
// Initialize Warrior constructor
function Warrior(name, level, weapon) {
  // Chain constructor with call
  Hero.call(this, name, level);

  // Add a new property
  this.weapon = weapon;
}

// Initialize Healer constructor
function Healer(name, level, spell) {
  Hero.call(this, name, level);

  this.spell = spell;
}
```

Both new constructors now have the properties of `Hero` and a few unqiue ones. We'll add the `attack()` method to `Warrior`, and the `heal()` method to `Healer`.

characterSelect.js

```
...
Warrior.prototype.attack = function () {
  return `${this.name} attacks with the ${this.weapon}.`;
}
```

```
Healer.prototype.heal = function () {
  return `${this.name} casts ${this.spell}.`;
}
```

At this point, we'll create our characters with the two new character classes available.

characterSelect.js

```
const hero1 = new Warrior('Bjorn', 1, 'axe');
const hero2 = new Healer('Kanin', 1, 'cure');
```

`hero1` is now recognized as a `Warrior` with the new properties.

```
Output
Warrior {name: "Bjorn", level: 1, weapon: "axe"}
```

We can use the new methods we set on the `Warrior` prototype.

```
hero1.attack();
```

```
Console
"Bjorn attacks with the axe."
```

But what happens if we try to use methods further down the prototype chain?

```
hero1.greet();
```

```
Output
Uncaught TypeError: hero1.greet is not a function
```

Prototype properties and methods are not automatically linked when you use `call()` to chain constructors. We will use `Object.create()` to link the prototypes, making sure to put it before any additional methods are created and added to the prototype.

characterSelect.js

```
...
Warrior.prototype = Object.create(Hero.prototype);
Healer.prototype = Object.create(Hero.prototype);

// All other prototype methods added below
...
```

Now we can successfully use prototype methods from `Hero` on an instance of a `Warrior` or `Healer`.

```
hero1.greet();
```

```
Output
"Bjorn says hello."
```

Here is the full code for our character creation page.

characterSelect.js

```
// Initialize constructor functions
function Hero(name, level) {
  this.name = name;
  this.level = level;
}

function Warrior(name, level, weapon) {
  Hero.call(this, name, level);

  this.weapon = weapon;
```

```
}

function Healer(name, level, spell) {
  Hero.call(this, name, level);

  this.spell = spell;
}

// Link prototypes and add prototype methods
Warrior.prototype = Object.create(Hero.prototype);
Healer.prototype = Object.create(Hero.prototype);

Hero.prototype.greet = function () {
  return `${this.name} says hello.`;
}

Warrior.prototype.attack = function () {
  return `${this.name} attacks with the ${this.weapon}.`;
}

Healer.prototype.heal = function () {
  return `${this.name} casts ${this.spell}.`;
}

// Initialize individual character instances
const hero1 = new Warrior('Bjorn', 1, 'axe');
const hero2 = new Healer('Kanin', 1, 'cure');
```

With this code we've created our `Hero` class with the base properties, created two character classes called `Warrior` and `Healer` from the original constructor, added methods to the prototypes and created individual character instances.

Conclusion

JavaScript is a prototype-based language, and functions differently than the traditional class-based paradigm that many other object-oriented languages use.

In this tutorial, we learned how prototypes work in JavaScript, and how to link object properties and methods via the hidden `[[Prototype]]` property that all objects share. We also learned how to create custom constructor functions and how prototype inheritance works to pass down property and method values.

By: Tania Rascia

Editor:
Lisa Tagliaferri

Tutorial Series

How To Code in JavaScript

JavaScript is a high-level, object-based, dynamic scripting language popular as a tool for making webpages interactive.

Show Tutorials