Tip 39: Extend Existing Prototypes with Class

In this tip, you'll learn how to use classes with existing prototypes.

We'll cover the following

Now that you know how to write classes in JavaScript, it's time to see how the new class syntax relates to JavaScript prototypes. It's important to understand that classes in JavaScript and prototypes aren't different. Classes are just a clean way to write regular JavaScript. By understanding how classes in JavaScript differ from traditional object-oriented languages, you'll be able to integrate new syntax with legacy code and prevent subtle bugs from surfacing.

Differences between JavaScript & other object-oriented languages

What are the differences between JavaScript and more traditional object-oriented languages? Here are the basics: When you use a class in traditional object-oriented languages, such as Ruby, it's a blueprint for an object. When you create a new instance, you copy all the properties and methods onto the new object.

JavaScript is a *prototype language*. When you create a new instance, you aren't copying methods. You're creating a *link* to a prototype. When you call a method on an instance of an object, you're calling it from the *prototype*, which is itself an object instance (not a blueprint). Eric Elliot has a longer article on the subject.

When you see the word *class* in JavaScript, you should know that it isn't new functionality. It's just a shorthand for a *prototype*. That means you can integrate class syntax with your current code bases.

Constructor functions

Up to this point, you've created object instances from classes, but not from constructor functions. In pre-ES5 JavaScript, when you wanted to create a new object instance using the `new` keyword, you'd use a function. You'll notice that *constructor functions* are very similar to a `constructor` method on a class. That should be a clue that new syntax will fit in nicely with legacy code.

To make an object instance with a constructor function in JavaScript, you'd simply write a function as normal. By convention, when you intend to use a function as a constructor, you'd start the function with a capital letter. Inside the function, you can attach properties to an instance using the `this` keyword.

When you create a new instance using the `new` keyword, you run the function as a constructor and bind the `this` context.

Example

Here's `Coupon` written as a constructor function.

```
function Coupon(price, expiration) {
    this.price = price;
    this.expiration = expiration || 'two weeks';
}
const coupon = new Coupon(5, 'two months');
console.log(coupon.price);
```

That should look familiar. All you did is pull out your constructor function into a standalone action. The only problem is you lost all your methods. This is precisely where JavaScript diverges from traditional object-oriented languages.

Prototype

When you created a new instance with `new`, you ran the constructor and bound a `this` context, but you didn't copy methods. You can add methods to this in the constructor, but it's far more efficient to add directly to a *prototype*.

A **prototype** is an object that's the *base* for the constructor function. All object instances *derive* properties from the prototype. In addition, new instances can also use methods on the prototype.

Adding methods to a prototype

To add a method to a prototype, you use the constructor name, `Coupon`, and you add the method to the prototype property as if you were adding a function or property to an object instance. Add the `getExpirationMessage()` method to the prototype. Now remember, you already have a working instance of `Coupon`. Because you're working with an instance of a prototype, you can access a method you add even after you've created a new instance.

```
function Coupon(price, expiration) {
    this.price = price;
    this.expiration = expiration || 'two weeks';
}

Coupon.prototype.getExpirationMessage = function () {
    return `This offer expires in ${this.expiration}.`;
};
const coupon = new Coupon(5, 'two months');
console.log(coupon.getExpirationMessage());
```

When you create an object using the `class` keyword, you're still creating prototypes and binding contexts, but with a more intuitive interface.

Extending a prototype

The code you just created using constructor functions and prototypes is identical to the classes you created in previous tips. It looks different, but behind the scenes, you're still creating a prototype.

And because they're the same, you can write classes for legacy code that you built using prototypes. For example, if you wanted to extend the `Coupon` prototype, the process would be the exact same as when you extended the `Coupon` you built with class syntax. You merely declare that you're extending the `Coupon` prototype when you create your new class.

```
class FlashCoupon extends Coupon {
    constructor(price, expiration) {
        super(price);
        this.expiration = expiration || 'two hours';
    }
    getExpirationMessage() {
        return `This is a flash offer and expires in ${this.expiration}.`;
    }
}


const flash = new FlashCoupon(5);
console.log(flash.price);
console.log(flash.getExpirationMessage());
```

If you spend lots of time with JavaScript, it’s worth exploring other ideas such as the prototypal chain, but for now, all you need to know is that classes aren’t new functionality. It’s just a new name for an old concept. Check out the Mozilla Developer Network for a few more examples of how classes relate to prototypes.

Q

Study the code below:

```
1- function Person(name, age) {
2-     this.name = name;
3-     this.age = age;
4- }
5-
6- Person.getName = function () {
7-     return `${this.name}.`;
8- };
```

Line 6 adds the `getName` function to the prototype of `Person`. True or False?

Retake Quiz

In the next tip, you’ll return to class syntax and explore how to make simple interfaces using get and set.