**Object-oriented JavaScript**

**Classes in JavaScript: Constructor Functions**

JavaScript does not have a traditional class system like Ruby does. Instead of declaring classes in JS, you instead define a constructor function:

function Kitten(name, age) {

this.name = name;

this.age = age;

this.meow = function () {

console.log(this.name + ' says "meow!"');

};

}

let kitten = new Kitten("Earl", 2);

Invoking a function with the special new keyword calls the function with this set to a new blank object. This blank object will be returned by the constructor as the new Kitten instance.

We can set attributes inside the constructor with this.key = value. Note that we don't declare any local variables; those would be temporary and thrown away at the end of construction. Instead, we persist data by saving it to object attributes using this.

Notice that Kitten **does not return anything**. Like initialize in Ruby, *constructor functions* in JS shouldn't return a value.

Because constructor functions are invoked differently than normal functions, make sure to name constructor functions in capitalized CamelCase to distinguish them clearly from normal functions. A common mistake is to call constructor functions without using new. If you do that, this will not be set properly, and no new object will be created.

**Methods and Prototype**

Our previous example added a meow method to our Kitten object by saving a function to the meow attribute of the newly constructed Kitten object. This is OK, but each time you build a Kitten object, you'll create a new function object for that kitten's meow.

This is redundant; all Kitten objects should share a single meow method. First, it will take more memory as we create more Kittens. Secondly, if we want to change our method, we have to update all our objects because they will individually have out-of-date copies.

We can eliminate the redundancy by using the constructor's prototype.

function Kitten(name, age) {

this.name = name;

this.age = age;

}

Kitten.prototype.meow = function () {

console.log(this.name + ' says "meow!"');

};

k1 = new Kitten("Earl", 2);

k2 = new Kitten("Houdini", 1);

Every function object has an attribute, prototype. We set instance methods on the prototype.

Whenever we construct a new Kitten instance, a special property kitten.\_\_proto\_\_ is set to Kitten.prototype. This links instances of the Kitten class to the Kitten.prototype that stores the instance methods.

k1 = new Kitten("Earl", 2);

k2 = new Kitten("Houdini", 1);

// `Object.getPrototypeOf` is the portable, preferred way to access

// the `\_\_proto\_\_` property.

Object.getPrototypeOf(k1); // == Kitten.prototype

Object.getPrototypeOf(k2); // == Kitten.prototype

Later, we may try to access a property of a Kitten instance (kitten.meow, for instance). JavaScript first looks for meow amongst the instance's own properties. In our example, these are name and age.

Since it doesn't find the meow property in the Kitten instance, the rule is that JavaScript will search in kitten.\_\_proto\_\_. Here we do have a value for the meow property, so the search is complete and the value is returned.

Let's see this illustrated a bit more:

// error, purr is not defined!

k1.purr();

Kitten.prototype.purr = function () {

console.log("Purr on, kitten!");

};

// all of a sudden it works! Because `Kitten.prototype` now has a

// `purr` property, `k1` can purr via its `k1.\_\_proto\_\_` reference to

// `Kitten.prototype`.

k1.purr();

In this way, **even though the**meow**method is only defined once,** because it is stored in Kitten.prototype, **every**Kitten**instance can access it** via the \_\_proto\_\_ property.

**Class Methods**

We can mimic Ruby class methods and class variables by defining attributes on the constructor itself.

Kitten.caboodle = [k1, k2, new Cat('Flying Merkel', 3)];

Kitten.parade = function() {

Kitten.caboodle.forEach( kitten => {

kitten.meow();

});

};

Kitten.parade();

// 'Earl says "meow!"'

// 'Houdini says "meow!"'

// 'Flying Merkel says "meow!"'

**Constructor Steps: Recap**

Say we call let cat = new Cat(). Here's what JS does:

1. JavaScript creates a new blank object.
2. JavaScript sets a special cat.\_\_proto\_\_ property to Cat.prototype. This way the instance of the class is connected to a prototype where all the instance methods are defined.
3. JavaScript runs the code in the body of the Cat function. It sets this to the blank object. The constructor presumably sets some attributes of the object.
4. JavaScript ignores the return value of the constructor function. Instead, the new Cat instance is returned, and set to cat.

**Method calling: Recap**

Later, when a cat.meow() is requested

1. JavaScript looks in the cat object for the meow property.
2. It doesn't find it, so it accesses the cat.\_\_proto\_\_ property.
3. This references Cat.prototype, JS searches for meow in the prototype.
4. Cat.prototype.meow is defined, so this is returned.
5. The meow method is called method style on cat, so this is set to cat