**Introduction to Classes**

JavaScript is an *object-oriented programming*(OOP) language we can use to model real-world items. In this lesson, you will learn how to make *classes*. Classes are a tool that developers use to quickly produce similar objects.

Take, for example, an object representing a dog named halley. This dog's name (a key) is "Halley" (a value) and has an age (another key) of 3 (another value). We create the halleyobject below:

let halley = { \_name: 'Halley', \_behavior: 0, get name() { return this.\_name; }, get behavior() { return this.\_behavior; }, incrementBehavior() { this.\_behavior++; } }

Now, imagine you own a dog daycare and want to create a catalog of all the dogs who belong to the daycare. Instead of using the syntax above for every dog that joins the daycare, we can create a Dog class that serves as a template for creating new Dog objects. For each new dog, you can provide a value for their name.

As you can see, classes are a great way to reduce duplicate code and debugging time.

After we lay the foundation for classes in the first few exercises, we will introduce inheritance and static methods — two features that will make your code more efficient and meaningful.

# Constructor

In the last exercise, you created a class called Dog, and used it to produce a Dog object.

Although you may see similarities between class and object syntax, there is one important method that sets them apart. It's called the constructor method. JavaScript calls the constructor() method every time it creates a new instance of a class.

class Dog { constructor(name) { this.name = name; this.behavior = 0; } }

* Dog is the name of our class. By convention, we capitalize and CamelCase class names.
* JavaScript will invoke the constructor()method every time we create a new instance of our Dog class.
* This constructor() method accepts one argument, name.
* Inside of the constructor() method, we use the this keyword. In the context of a class, this refers to an instance of that class. In the Dog class, we use this to set the value of the Dog instance's name property to the name argument.
* Under this.name, we create a property called behavior, which will keep track of the number of times a dog misbehaves. The behavior property is always initialized to zero.

In the next exercise, you will learn how to create Dog instances.

**Instance**

Now, we're ready to create class instances. An *instance* is an object that contains the property names and methods of a class, but with unique property values. Let's look at our Dog class example.

class Dog { constructor(name) { this.name = name; this.behavior = 0; } } const halley = new Dog('Halley'); // Create new Dog instance console.log(halley.name); // Log the name value saved to halley // Output: 'Halley'

Below our Dog class, we use the new keyword to create an instance of our Dog class. Let's consider the line of code step-by-step.

* We create a new variable named halleythat will store an instance of our Dog class.
* We use the new keyword to generate a new instance of the Dog class. The newkeyword calls the constructor(), runs the code inside of it, and then returns the new instance.
* We pass the 'Halley' string to the Dogconstructor, which sets the name property to 'Halley'.
* Finally, we log the value saved to the namekey in our halley object, which logs 'Halley' to the console.

Now you know how to create instances. In the next exercise, you will learn how to add getters, setters, and methods.

# Methods

At this point, we have a Dog class that spins up objects with name and behavior properties. Below, we will add getters and a method to bring our class to life.

Class method and getter syntax is the same as it is for objects **except you can not include commas between methods**.

class Dog { constructor(name) { this.\_name = name; this.\_behavior = 0; } get name() { return this.\_name; } get behavior() { return this.\_behavior; } incrementBehavior() { this.\_behavior++; } }

In the example above, we add getter methods for name and behavior. Notice, we also prepended our property names with underscores (\_name and \_behavior), which indicate these properties should not be accessed directly. Under the getters, we add a method named .incrementBehavior(). When you call .incrementBehavior() on a Dog instance, it adds 1 to the \_behavior property. Between each of our methods, we did notinclude commas.

# Method Calls

Finally, let's use our new methods to access and manipulate data from Dog instances.

class Dog { constructor(name) { this.\_name = name; this.\_behavior = 0; } get name() { return this.\_name; } get behavior() { return this.\_behavior; } incrementBehavior() { this.\_behavior++; } } const halley = new Dog('Halley');

In the example above, we create the Dog class, then create an instance, and save it to a variable named halley.

The syntax for calling methods and getters on an instance is the same as calling them on an object — append the instance with a period, then the property or method name. For methods, you must also include opening and closing parentheses.

Let's take a moment to create two Doginstances and call our .incrementBehavior()method on one of them.

let nikko = new Dog('Nikko'); // Create dog named Nikko nikko.incrementBehavior(); // Add 1 to nikko instance's behavior let bradford = new Dog('Bradford'); // Create dog name Bradford console.log(nikko.behavior); // Logs 1 to the console console.log(bradford.behavior); // Logs 0 to the console

In the example above, we create two new Doginstances, nikko and bradford. Because we increment the behavior of our nikko instance, but not bradford, accessing nikko.behaviorreturns 1 and accessing bradford.behaviorreturns 0.

# Inheritance I

Imagine our doggy daycare is so successful that we decide to expand the business and open a kitty daycare. Before the daycare opens, we need to create a Cat class so we can quickly generate Cat instances. We know that the properties in our Cat class (name, behavior) are similar to the properties in our Dog class, though, there will be some differences, because of course, cats are not dogs.

Let's say that our Cat class looks like this:

class Cat { constructor(name, usesLitter) { this.\_name = name; this.\_usesLitter = usesLitter; this.\_behavior = 0; } get name() { return this.\_name; } get usesLitter() { return this.\_usesLitter; } get behavior() { return this.\_behavior; } incrementBehavior() { this.\_behavior++; } }

In the example above, we create a Cat class. It shares a couple of properties (\_name and \_behavior) and a method (.incrementBehavior()) with the Dog class from earlier exercises. The Cat class also contains one additional property (\_usesLitter), that holds a boolean value to indicate whether a cat can use their litter box.

When multiple classes share properties or methods, they become candidates for inheritance — a tool developers use to decrease the amount of code they need to write.

With inheritance, you can create a parent class (also known as a superclass) with properties and methods that multiple child classes (also known as subclasses) share. The child classes inherit the properties and methods from their parent class.

Let's abstract the shared properties and methods from our Cat and Dog classes into a parent class called Animal.

class Animal { constructor(name) { this.\_name = name; this.\_behavior = 0; } get name() { return this.\_name; } get behavior() { return this.\_behavior; } incrementBehavior() { this.\_behavior++; } }

In the example above, the Animal class contains the properties and methods that the Cat and Dog classes share (name, behavior, .incrementBehavior()).

The diagram to the right shows the relationships we want to create between the Animal, Cat, and Dog classes.

# Inheritance II

In the last exercise, we created a parent class named Animal for two child classes named Catand Dog.

The Animal class below contains the shared properties and methods of Cat and Dog.

class Animal { constructor(name) { this.\_name = name; this.\_behavior = 0; } get name() { return this.\_name; } get behavior() { return this.\_behavior; } incrementBehavior() { this.\_behavior++; } }

The code below shows the Cat class that will inherit information from the Animal class.

class Cat { constructor(name, usesLitter) { this.\_name = name; this.\_usesLitter = usesLitter; this.\_behavior = 0; } get name() { return this.\_name; } get behavior() { return this.\_behavior; } get usesLitter() { return this.\_usesLitter; } incrementBehavior() { this.\_behavior++; } }

To the right, in **main.js**, you will put what you learned to practice by creating a parent class named HospitalEmployee.

**Inheritance III**

We've abstracted the shared properties and methods of our Cat and Dog classes into a parent class called Animal (See below).

class Animal { constructor(name) { this.\_name = name; this.\_behavior = 0; } get name() { return this.\_name; } get behavior() { return this.\_behavior; } incrementBehavior() { this.\_behavior++; } }

Now that we have these shared properties and methods in the parent Animal class, we can extend them to the subclass, Cat.

class Cat extends Animal { constructor(name, usesLitter) { super(name); this.\_usesLitter = usesLitter; } }

In the example above, we create a new class named Cat that extends the Animal class. Let's pay special attention to our new keywords: extends and super.

* The extends keyword makes the methods of the animal class available inside the cat class.
* The constructor, called when you create a new Cat object, accepts two arguments, name and usesLitter.
* The super keyword calls the constructor of the parent class. In this case, super(name)passes the name argument of the Cat class to the constructor of the Animal class. When the Animal constructor runs, it sets this.\_name = name; for new Cat instances.
* \_usesLitter is a new property that is unique to the Cat class, so we set it in the Cat constructor.

Notice, we call super on the first line of our constructor(), then set the usesLitterproperty on the second line. In a constructor(), you must always call the super method before you can use the this keyword — if you do not, JavaScript will throw a reference error. To avoid reference errors, it is best practice to call superon the first line of subclass constructors.

Below, we create a new Cat instance and call its name with the same syntax as we did with the Dog class:

const bryceCat = new Cat('Bryce', false); console.log(bryceCat.\_name); // output: Bryce

In the example above, we create a new instance the Cat class, named bryceCat. We pass it 'Bryce' and false for our name and usesLitter arguments. When we call console.log(bryceCat.\_name) our program prints, Bryce.

In the example above, we abandoned best practices by calling our \_name property directly. In the next exercise, we'll address this by calling an inherited getter method for our nameproperty.

**Inheritance IV**

Now that we know how to create an object that inherits properties from a parent class let's turn our attention to methods.

When we call extends in a class declaration, all of the parent methods are available to the child class.

Below, we extend our Animal class to a Catsubclass.

class Animal { constructor(name) { this.\_name = name; this.\_behavior = 0; } get name() { return this.\_name; } get behavior() { return this.\_behavior; } incrementBehavior() { this.\_behavior++; } } class Cat extends Animal { constructor(name, usesLitter) { super(name); this.\_usesLitter = usesLitter; } } const bryceCat = new Cat('Bryce', false);

In the example above, our Cat class extends Animal. As a result, the Cat class has access to the Animal getters and the .incrementBehavior() method.

Also in the code above, we create a Catinstance named bryceCat. Because bryceCathas access to the name getter, the code below logs 'Bryce' to the console.

console.log(bryceCat.name);

Since the extends keyword brings all of the parent's getters and methods into the child class, bryceCat.name accesses the name getter and returns the value saved to the nameproperty.

Now consider a more involved example and try to answer the following question: What will the code below log to the console?

bryceCat.incrementBehavior(); // Call .incrementBehavior() on Cat instance console.log(bryceCat.behavior); // Log value saved to behavior

The correct answer is 1. But why?

* The Cat class inherits the \_behaviorproperty, behavior getter, and the .incrementBehavior() method from the Animal class.
* When we created the bryceCat instance, the Animal constructor set the \_behaviorproperty to zero.
* The first line of code calls the inherited .incrementBehavior() method, which increases the bryceCat \_behavior value from zero to one.
* The second line of code calls the behaviorgetter and logs the value saved to \_behavior (1).

# Inheritance V

In addition to the inherited features, child classes can contain their own properties, getters, setters, and methods.

Below, we will add a usesLitter getter. The syntax for creating getters, setters, and methods is the same as it is in any other class.

class Cat extends Animal { constructor(name, usesLitter) { super(name); this.\_usesLitter = usesLitter; } get usesLitter() { return this.\_usesLitter; } }

In the example above, we create a usesLittergetter in the Cat class that returns the value saved to \_usesLitter.

Compare the Cat class above to the one we created without inheritance:

class Cat { constructor(name, usesLitter) { this.\_name = name; this.\_usesLitter = usesLitter; this.\_behavior = 0; } get name() { return this.\_name; } get usesLitter() { return this.\_usesLitter; } get behavior() { return this.\_behavior; } incrementBehavior() { this.\_behavior++; } }

We decreased the number of lines required to create the Cat class by about half. Yes, it did require an extra class (Animal), making the reduction in the size of our Cat class seem moot. However, the benefits (time saved, readability, efficiency) of inheritance grow as the number and size of your subclasses increase.

One benefit is that when you need to change a method or property that multiple classes share, you can change the parent class, instead of each subclass.

Before we move past inheritance, take a moment to see how we would create an additional subclass, called Dog.

class Dog extends Animal { constructor(name) { super(name); } }

This Dog class has access to the same properties, getters, setters, and methods as the Dog class we made without inheritance, and is a quarter the size.

Now that we've abstracted animal daycare features, it's easy to see how you can extend Animal to support other classes, like Rabbit, Bird or even Snake.

# Static Methods

Sometimes you will want a class to have methods that aren't available in individual instances, but that you can call directly from the class.

Take the Date class, for example — you can both create Date instances to represent whatever date you want, and call staticmethods, like Date.now() which returns the current date, directly from the class. The .now()method is static, so you can call it directly from the class, but not from an instance of the class.

Let's see how to use the static keyword to create a static method called generateNamemethod in our Animal class:

class Animal { constructor(name) { this.\_name = name; this.\_behavior = 0; } static generateName() { const names = ['Angel', 'Spike', 'Buffy', 'Willow', 'Tara']; const randomNumber = Math.floor(Math.random()\*5); return names[randomNumber]; } }

In the example above, we create a staticmethod called .generateName() that returns a random name when it's called. Because of the static keyword, we can only access .generateName() by appending it to the Animalclass.

We call the .generateName() method with the following syntax:

console.log(Animal.generateName()); // returns a name

You cannot access the .generateName() method from instances of the Animal class or instances of its subclasses (See below).

const tyson = new Animal('Tyson'); tyson.generateName(); // TypeError

The example above will result in an error, because you cannot call static methods (.generateName()) on an instance (tyson).

**Review: Classes**

Way to go! Let's review what you learned.

* *Classes* are templates for objects.
* Javascript calls a *constructor* method when we create a new instance of a class.
* *Inheritance* is when we create a parent class with properties and methods that we can extend to child classes.
* We use the extends keyword to create a subclass.
* The super keyword calls the constructor()of a parent class.
* Static methods are called on the class, but not on instances of the class.

In completing this lesson, you've taken one step closer to writing efficient, production-level JavaScript. Good luck as you continue to develop your skills and move into intermediate-level concepts.