Create a Basic JavaScript Object

Think about things people see every day, like cars, shops, and birds. These are all *objects*: tangible things people can observe and interact with.

What are some qualities of these objects? A car has wheels. Shops sell items. Birds have wings.

These qualities, or *properties*, define what makes up an object. Note that similar objects share the same properties, but may have different values for those properties. For example, all cars have wheels, but not all cars have the same number of wheels.

Objects in JavaScript are used to model real-world objects, giving them properties and behavior just like their real-world counterparts. Here's an example using these concepts to create a `duck` object:

```
let duck = {
  name: "Aflac",
  numLegs: 2
};
```

This `duck` object has two property/value pairs: a `name` of "Aflac" and a `numLegs` of 2.

Use Dot Notation to Access the Properties of an Object

The last challenge created an object with various properties. Now you'll see how to access the values of those properties. Here's an example:

```
let duck = {
  name: "Aflac",
  numLegs: 2
};
console.log(duck.name);
// This prints "Aflac" to the console
```

Dot notation is used on the object name, `duck`, followed by the name of the property, `name`, to access the value of "Aflac".

Create a Method on an Object

Objects can have a special type of property, called a *method*.

Methods are properties that are functions. This adds different behavior to an object. Here is the `duck` example with a method:

```
let duck = {
  name: "Aflac",
  numLegs: 2,
  sayName: function() {return "The name of this duck is " + duck.name
+ ".";}
};
duck.sayName();
// Returns "The name of this duck is Aflac."
```

The example adds the `sayName` method, which is a function that returns a sentence giving the name of the `duck`. Notice that the method accessed the `name` property in the return statement using `duck.name`. The next challenge will cover another way to do this.

Make Code More Reusable with the this Keyword

The last challenge introduced a method to the `duck` object. It used `duck.name` dot notation to access the value for the `name` property within the return statement:

```
sayName: function() {return "The name of this duck is " + duck.name +
".";}
```

While this is a valid way to access the object's property, there is a pitfall here. If the variable name changes, any code referencing the original name would need to be updated as well. In a short object definition, it isn't a problem, but if an object has many references to its properties there is a greater chance for error.

A way to avoid these issues is with the `this` keyword:

```
let duck = {
  name: "Aflac",
  numLegs: 2,
  sayName: function() {return "The name of this duck is " +
this.name + ".";}
};
```

`this` is a deep topic, and the above example is only one way to use it. In the current context, `this` refers to the object that the method is associated with: `duck`. If the object's name is changed to `mallard`, it is not necessary to find all the references to `duck` in the code. It makes the code reusable and easier to read.

Define a Constructor Function

Constructors are functions that create new objects. They define properties and behaviors that will belong to the new object. Think of them as a blueprint for the creation of new objects.

Here is an example of a constructor:

```
function Bird() {
  this.name = "Albert";
  this.color = "blue";
  this.numLegs = 2;
}
```

This constructor defines a `Bird` object with properties `name`, `color`, and `numLegs` set to Albert, blue, and 2, respectively. Constructors follow a few conventions:

- Constructors are defined with a capitalized name to distinguish them from other functions that are not `constructors`.
- Constructors use the keyword `this` to set properties of the object they will create. Inside the constructor, `this` refers to the new object it will create.
- Constructors define properties and behaviors instead of returning a value as other functions might.

Use a Constructor to Create Objects

Here's the `Bird` constructor from the previous challenge:

```
function Bird() {
  this.name = "Albert";
  this.color  = "blue";
  this.numLegs = 2;
  // "this" inside the constructor always refers to the object being
created
}


let blueBird = new Bird();
```

Notice that the `new` operator is used when calling a constructor. This tells JavaScript to create a new instance of `Bird` called `blueBird`. Without the `new` operator, `this` inside the constructor would not point to the newly created object, giving unexpected results. Now `blueBird` has all the properties defined inside the `Bird` constructor:

```
blueBird.name; // => Albert
blueBird.color; // => blue
blueBird.numLegs; // => 2
```

Just like any other object, its properties can be accessed and modified:

```
blueBird.name = 'Elvira';
blueBird.name; // => Elvira
```