Monday: Constructors and Prototypes

Now, let's explore how JavaScript uses constructors as blueprints for the creation of many new objects, and prototypes for adding methods to objects.

Let's start by taking a look at how some of the built-in JavaScript objects work.

Constructors

If we look at the **MDN documentation for `String` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)**, we see that `String` is a **constructor** for creating string objects. A constructor is a function that can be invoked using the `new` keyword to create new objects. To visually identify constructor functions, they are conventionally named with a capital first letter.

Previously, we have created strings simply by adding `" "` around the characters we want to identify as strings. This is called **literal notation**. It triggers JavaScript to construct a string object with the String constructor. But we could also use the `String` constructor function directly:

```
> var testGreeting = new String;
  undefined

> testGreeting
  String {length: 0, [[PrimitiveValue]]: ""}
```

Or this:

```
> var testGreeting2 = new String("Hello!");
  undefined

> testGreeting2
  String {0: "H", 1: "e", 2: "l", 3: "l", 4: "o", 5: "!", length: 6, [[PrimitiveValue]]: "Hell
o!"}
```

In these two examples, we see the `String` constructor is called with the `new` keyword. The `String` constructor function added 2 properties to both objects - `length` and `[[PrimitiveValue]]`.
For `testGreeting2`, where we provided a value as the argument, the constructor also added properties for the index positions of each character.

In the case of creating a new String, it was actually easier to use the literal notation. But with more complex objects it will often be easier to use constructors when creating new objects that all have the same blueprint.

Let's make a constructor of our own. We'll imagine that we need to create many dogs from a dog blueprint since we know that all dogs will have the same properties. The difference from dog to dog will be in the values for those properties. Rather than repeating all of the code for each dog, we'll make a constructor function that we can use over and over again.

Here is a constructor function for `Dog` that will initialize a new dog object with its attributes assigned to the values passed into the constructor function.

```
function Dog(name, colors, age) {
  this.name = name;
  this.colors = colors;
  this.age = age;
}
```

Then to create a new dog we can do the following:

```
var myPuppy = new Dog("Ernie", ["brown","black"], 3);
```

We can access the name of the new dog:

```
> myPuppy.name;
  "Ernie"
```

The colors of the new dog:

```
> myPuppy.colors;
  ["brown","black"]
```

And its age:

```
> myPuppy.age;
```

The `myPuppy` object here is considered an **instance** of the `Dog` type. A constructor is the blueprint that specifies how to create an object. You can think of the `Dog` constructor here as a factory that can be used repeatedly to build a bunch of dog objects, using the constructor as a blueprint. Each dog object is an *instance* of the `Dog` type, so you have one `Dog` type, defined by a constructor, with potentially many instances of that type.

```
function Dog(name, colors, age) {
  this.name = name;
  this.colors = colors;
  this.age = age;
}

var falcor = new Dog("Falcor", ["black"], 4);
var nola = new Dog("Nola", ["white", "black"], 6);
var patsy = new Dog("Patsy", ["brown"], 7);
```

Just to reiterate, the `Dog` constructor above is used as a kind of factory or blueprint by each of the three lines below in order to construct the three dog objects. The `falcor`, `nola` and `patsy` objects are three instances of the `Dog` type. Each object that is an instance of `Dog` has a `name` property, a `colors` property and an `age` property. For example:

```
> falcor.name
  "Falcor"
> nola.name
  "Nola"
> nola.age
  6
> patsy.colors
  ["brown"]
```

Now we know how to create new objects with properties, but what about an object's methods? Let's take a look at methods for `String` and `Dog` next.

Prototypes

We know that JavaScript also has a number of built-in string methods for each new String created. When we show our `testGreeting2`, those methods are not listed in the curly braces. This is because the constructor function is only used to create a new object's properties.

To define methods, JavaScript employs **prototypes**. If we look at the methods for `String` on **MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)** again, we see that they are all listed with the notation, String.prototype. *A prototype is just an object from which other objects inherit methods*. So, all instances of the String constructor inherit from the String.prototype (just as all instances of Dog will inherit from Dog.prototype). When you call a method, JavaScript first looks at the methods on the object, and if it doesn't find them there, it looks to the prototype.

Let's take a look at how this works with our `testGreeting2` string object.

We can run any of the String.prototype methods on the string `testGreeting2` which currently has a value of "Hello!". Let's try `.toUpperCase()`.

```
> testGreeting2.toUpperCase();
  "HELLO!"
```

When we execute this, JavaScript first searches the properties of the `testGreeting2` object. It doesn't find it. It then goes to the String.prototype object, finds the `toUpperCase` method there, and processes the function as requested.

Let's add a custom method to String.prototype:

```
> String.prototype.addExcitement = function() { return this + "!!!!!!!!!!" };

> testGreeting2.addExcitement();
  "Hello!!!!!!!!!!"
```

As soon as the new method is added, all current and future instances of String will have access to it. Now I can run `testGreeting2.addExcitement()` and get `Hello!!!!!!!!!`.

If I create a new string, it, too, will have access to the prototype's `.addExcitement()` method:

```
> var newGreeting = "Jambo";
  undefined

> newGreeting.addExcitement();
  "Jambo!!!!!!!!!"
```

We might think, why wouldn't methods just be added to the constructor instead of having a separate prototype object? If all methods were added to the constructor, EVERY new object would create additional function objects for EVERY method. By adding them to a shared prototype, the function objects are created once and shared by all of the instances, which is more efficient.

Let's look at our Dog again. We can add our original methods to the Dog.prototype so that all dogs have these behaviors available to them.

```
Dog.prototype.speak = function() {
  console.log("Woof!");
}

Dog.prototype.humanYears = function() {
  return this.age * 7;
}
```

`myPuppy` can now speak:

```
> myPuppy.speak();
  Woof!
```

and have its age calculated in human years:

```
> myPuppy.humanYears();
  21
```

Every new dog will also have these methods.

```
>var newPuppy = new Dog("Goliath", ["gray"], 2);

> newPuppy.speak();
  Woof!
```

```
> newPuppy.humanYears();
  14
```

In summary, every time we create a new dog using the `new` keyword, it calls the `Dog` constructor, which provides the blueprint for creating an instance of the `Dog` type, giving it certain properties. The new instance of the `Dog` type also automatically gains access to all methods defined on the shared `Dog` prototype.