Constructor Functions

(https://github.com/learn-co-curriculum/phase-1-constructor-functions) (https://github.com/learn-co-curriculum/phase-1-constructor-functions/issues/new)

Learning Goals

- Create a constructor function
- Use a constructor function to create an object
- Explain what a constructor function is and how it works
- Explain what `new` is and how it works with the constructor function

Introduction

JavaScript is sometimes a bit confusing. It has a native type `Object`. How does that relate to Object-Orientation? And how does the desire to avoid repeating ourselves relate to them both? In this lesson we'll work with plain old JavaScript `Object`s, and slowly transition to one of the key elements of OO, the constructor function and its partner, `new`.

Create a Constructor Function

It's sometime handy to represent data with objects, which gives us key/value pairs. For example, we may represent a user as the following:

```
const guadalupe = { name: "guadalupe", age: 34, hometown: "Denver" };
```

Now imagine that we had a couple of users:

```
const guadalupe = {
  name: "guadalupe",
  age: 34,
  hometown: "Denver",
```

```
};

const aki = {
  name: "aki",
  age: 58,
  hometown: "Los Angeles",
};
```

Great. Two nice users.

Note, that with both objects sharing exactly the same keys, and only the values differing, we are ***repeating ourselves*** ***(https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)***. We would like a mechanism to construct objects with the same attributes (that is, keys), while assigning different values to those keys. The name for what kind of function does this varies across many popular programming languages, but we'll call it **a factory function** because it spits out new instances.

```
function User(name, age, hometown) {
  return {
    name, // don't forget ES6 power-tools, this is the same as `name: name`
    age,
    hometown,
  };
}

const amaru = User("Amaru", 45, "New York City");
amaru.age; // => 45
```

Interestingly `typeof` confirms `amaru` is an `Object`:

```
typeof amaru;
// => 'object'
```

However, something's not quite as clear as we might like it to be. If we ask `amaru` what made it, the answer is...

```
amaru.constructor;
// => [Function: Object]
```

`amaru` is certainly an `Object` but it's more specific than that: it's a `User`. We'd really like for this special kind of object to be reflected when we ask it what it is. We'd like for a mystical process to come along and say you are not merely an `Object`, you are a `User` or a `Dog`. We tell JavaScript to bless the thing created by the constructor function into being something more specific than `Object` by using the keyword `new`. Using `new` requires that we evolve our *factory function* into a *constructor function*. It's the same idea, but with a few subtle additions.

Explain What `new` Is and How It Works With the Constructor Function

Lets create a *constructor function*. Constructor functions must be paired with the `new` keyword (which we'll cover in a moment).

```
function User(name, email) {
  this.name = name;
  this.email = email;
}
```

What's `this` here? It refers to the function's context. Since functions in JavaScript are also `Object`s, a function can say "on me, set a property." As we read this function we might think "OK, so you're going to run a function that will set properties on itself. That's not doing anything useful."

We would be right.

What we really want to say is something like "Hey, constructor function, when you run, create a new copy of yourself, leaving the original unchanged and on that *particular* copy, set the properties based on the arguments passed into the function. The keyword `new` tells the constructor function to do exactly that.

Put the two together like so:

```
function User(name, email) {
  this.name = name;
  this.email = email;
```

```
}

const greyson = new User("Greyson", "greyson@example.com");
greyson.name; //=> "Greyson"
```

Remember you can and should try these out for yourself in the JavaScript console or in the `node` REPL

What's happening here hinges on the `new` keyword. When we invoke the function with `new` before it, you can imagine an imaginary JavaScript `Object` being copied for use in the `User` function. The constructor function **is not changed**; the freshly-created, new context **is** changed.

Here's the code sample above, but with some more comments. Follow the [numbers] 1-6.

```
function User(name, email) {
  this.name = name; // [2] Set my context's property name to what
  //     came in in the first argument (name)
  this.email = email; // [3] Set my context's property email to what
  //     came in in the second argument (email)
}

// [1] Create a new "context", that's what `new` does
// Use _that_ new context inside of the execution of the `User` function
// also pass two parameters, "Greyson" and "greyson@example.com"

// [4]: Assign the new context thing with its this properties set to the
// variable `greyson`
const greyson = new User("Greyson", "greyson@example.com");

// [5]: Ask the new context for what's in its `.name` property
greyson.name; //=> [6] "Greyson"
```

You can ask interesting questions about the `greyson` variable. Building on the previous code:

```
typeof greyson;
// => 'object'
greyson.constructor;
// => [Function: User]
```

This sorta makes sense, the function that constructed `greyson`, or the `constructor` is `User`. The instance `greyson` is an `object`. Given what we know about the types available in JavaScript, `object` makes good sense (not a `Number` or a `String`, that's for sure!)

Since we know one OO pattern, we might be wondering how to add methods to the `User` instances. It should be obvious that if we can set a property to point to a value like `"greyson"` or `"greyson@example.com"`, we should be able to set an anonymous function to a property. That function would have access to the `this` context created when the instance was `new`'d into existence.

```
function User(name, email) {
  this.name = name;
  this.email = email;
  this.sayHello = function () {
    console.log(`Hello everybody, my name is ${this.name} whom you've been
mailing at ${this.email}!`);
  };
}

const greyson = new User("Greyson", "greyson@example.com");
greyson.sayHello();
// => Hello everybody, my name is Greyson whom you've been mailing at greyson@example.com!
```

Feel free to try it with other names and emails to prove to yourself that you can create instances from this constructor function.

Conclusion

In this lesson you've seen the constructor function and the `new` keyword work to create new instances with instance data and instance methods. We've pretty much got an object oriented language at our fingers! Not bad!

There's one problem with this design. It's incredibly memory inefficient. Take a look at the code and imagine what that inefficiency could be. We'll explore it together in the next lesson!