Objects

WE'LL COVER THE FOLLOWING

Objects have received some new updates in this edition of the language, including everything from new methods to updated syntax.

Shorthand methods

One nice addition to objects is the shorthand method syntax. We will see this pattern come up again when we talk about classes. As a reminder, here is what the original syntax looks like.

```
let flower = {
    height: 10,
    colour: 'yellow',
    grow: function() {
        this.height += 5;
    }
}
```

Here we have a simple `flower` object which has a `height` and `colour` and a `grow` method. When defining a method we provide a function as a value to the property on an object. In this case we use an anonymous function. We could, if we wanted, also provide a named function.

```
let flower = {
    height: 10,
    colour: 'yellow',
    grow: function growMethod() {
        this.height += 5;
    }
}
```

The benefit of providing a named function is that we could use this name inside of our method if we need to call it recursively. With the new shorthand method syntax, we can exclude the anonymous function all together.

```
let flower = {
    height: 10,
    colour: 'yellow',
    grow() {
        this.height += 5;
    }
}
```

Look at that! I think this looks really nice.

Shorthand properties

Another new feature is the ability to create object properties via a shorthand syntax. There is not much to this, and we will see this being used later in the Destructuring chapter. Let’s consider the code below: assume we have some variables that we want to add to an object.

```
let firstName = "Ryan";
let lastName = "Christiani";

let person = { firstName, lastName };
```

The keys in our new object will be the names of the variables we provided it. The above is a shorthand for doing this:

```
let person = {
    firstName: firstName,
    lastName: lastName
};
```

It really is that simple; this new syntax allows us to create objects from existing variables with ease. A place this pattern is kind of nice is the revealing module pattern.

The idea of the revealing module pattern is that you have a function, that is acting as a module, that has data you want to keep private. From the module you return a public object that only reveals what you want to the user.

```
function slider() {
    let currentPosition = 0;
    function updatePosition(newPosition) {
        currentPosition = currentPostion + newPosition;
    }
    function resetPosition() {
        currentPosition = 0;
    }
    return {
        updatePostion,
        resetPostion
    };
}
```

Above we have a little slider module that doesn't actually do anything, but it illustrates the point. We have some private data, the `currentPosition` value, we also have two functions that will mutate that value. We expose or reveal the functions that we want, and with the shorthand property method we can make this nice and easy!

Computed properties

Computed properties is another new thing we can do with object properties in ES6. When creating an object, properties are made up of key value pairs. Typically a key could be a name like `firstName` or it could be wrapped in quotes `"firstName"`. With computed properties we are able to use expressions or existing variables to create keys.

```
let keyName = 'firstName';
let person = {
    [keyName]: 'Ryan'
};
```

As mentioned above, we can use expressions to create keys.

```
let person = {
    ['first' + 'name']: 'Ryan'
};
```

Object.assign()

The last thing involving objects that I want to go over is the `.assign()`

method. This new method objects allows us to create copies of existing objects, and also mix objects together.

The method is fairly straightforward. The first argument in the method is the target you want to assign, and any arguments after that are sources that you want to add.

As an example, let's think about a video game. We have some base objects, like a creature, and we have different versions of these creatures.

```
let bat = {
    weight: 10,
    strength: 4,
    altitude: 0,
    fly(newAltitude) {
        this.altitude += newAltitdude;
    }
};
```

We have our base object that has some simple stats, now say we have a greater bat!

```
let greaterBat = {
    weight: 15,
    strength: 7
};
```

The new bat has a few newer properties, but we also want it to be able to fly! There is a way to solve this, we could use inheritance via function prototypes, or we could use classes. We will look at classes later. For this example, we will favour composition of our object. Let's change `greaterBat` to `greaterBatProps` so we can use it in `Object.assign()`.

```
let greaterBatProps = {
    weight: 15,
    strength: 7
};

let greaterBat = Object.assign({}, bat, greaterBatProps);
console.log(greaterBat);
```

Using `Object.assign()` we are composing greaterBat into bat and finally into an empty object that acts as our target. The order of our arguments matter here, if we had called it like `Object.assign({},greaterBatProps,bat);` we would have a whole object. However because `bat` came last in this example, it would override anything in `greaterBatProps`. When you look at this read it from the right to the left as this is how it will compose!

This pattern is really great, and there is another great benefit to `Object.assign()` and that is to use it to copy objects! Consider the example below.

```
let person = {
    name: 'Ryan'
};

let person2 = person;

person2.name = "Erin";

console.log(person2.name); //Erin
console.log(person.name); //Erin
```

We have created a `person` object, and gave it a simple property. Then we create a new variable called `person2` and assign it `person`. You might think that this would add a copy of the person object, but a new person object.

However that is not the case, it just uses `person` as a reference, so changing the `name` property on `person2` is going to change it on `person`!

Using `Object.assign()` we can create a copy of `person` with ease!

```
let person = {
    name: 'Ryan'
};

let person2 = Object.assign({},person);

person2.name = "Erin";

console.log(person2.name); //Erin
console.log(person.name); //Ryan
```

Remember that the first argument to `Object.assign()` is the target, so using a blank object, we are creating a copy of the `person`.