Lesson Tuesday

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / ES6 Classes

Text

We've already covered ES6 support for modules, which are implemented with `import` and `export`. In this lesson, we'll go over ES6's native support for classes. Classes make JavaScript more accessible and easier to read from an object-oriented (OO) perspective. However, they don't fundamentally change the way JavaScript works.

`class` Syntax

Classes are a cornerstone of many OO languages, including Ruby, Java, and C#, but JavaScript didn't include support for classes until ES6. A class is used to define an object type. So far, we've used constructor functions to define object types, and added functionality to those objects via prototype methods. Using `class` syntax will do the exact same thing, just with different syntax.

Let's look at how we can refactor our `Triangle` object to use `class` syntax. To do this, we'll reference the business logic for `triangle.js` in our Shape Tracker project:

src/triangle.js

```
function Triangle(side1, side2, side3) {
  this.side1 = side1;
  this.side2 = side2;
  this.side3 = side3;
}

Triangle.prototype.checkType = function() {
  ...
};
```

Here we create a `Triangle` constructor and then a prototype method for that constructor. Let's update this code to use `class` syntax.

src/triangle.js

```
class Triangle {
  constructor(side1, side2, side3) {
    this.side1 = side1;
    this.side2 = side2;
    this.side3 = side3;
  }

  checkType() {
    //Function body goes here.
  }
}
```

Our `class Triangle` now contains both the `Triangle` constructor and all its prototype methods. The `Triangle()` constructor function now uses the `constructor()` function, and we no longer need to

specify the object type and `prototype` when we declare the `checkType()` method.

This code looks very similar to how we might construct a class in other OO languages such as Ruby and C#.

However, it's important to remember that JavaScript classes are syntactic sugar. **Syntactic sugar** is a term developers use for added functionality in a programming language that makes it easier to write and read.

JavaScript classes are syntactic sugar because they don't operate in the same way that classes do in other OO languages such as Ruby. The biggest difference is that JavaScript doesn't directly use **classical inheritance** where classes inherit functionality from other classes. Instead, JavaScript uses what's commonly referred to as **prototypal inheritance**, where object types inherit from other object types through a prototype object.

Inheritance in JavaScript

Classical inheritance simply means that one class inherits from another class. While classical inheritance has its advantages, it has one major disadvantage: when one class inherits from another, it inherits *everything*. The coder Joe Armstrong explains this problem with an apt metaphor:

> You wanted a banana, but what you got was a gorilla holding the banana and the entire jungle.

With **prototypal inheritance**, objects inherit from other objects through a prototype object. The prototype object is saved in the `__proto__` property on any JavaScript object, and it contains the functionality that it's inheriting from another object. Prototypal inheritance is an advanced topic beyond the scope of this lesson,

but what's important in comparison to classical inheritance is that the scope of prototypal inheritance is more limited. In other words, if you want a banana, you'll just get a banana.

The new ES6 `class` syntax fakes classical inheritance by building it on top of prototypal inheritance. In other words, we can use this functionality to have one class inherit from another. To draw this connection, we use the `extends` keyword:

```
class Shape {
  ...
}

class Triangle extends Shape {
  ...
}
```

You won't be expected to create one class that inherits from another for any code review, but you're welcome to explore inheritance further in your multi-day project.

However, you will be expected to use `class` syntax to create object types for the upcoming code review.

Using Variables in ES6 Classes

There is one important thing to note about ES6 `class` syntax. Variables cannot be scoped to the class itself. The following will not work:

```
class Triangle {
  let variableScopedToClass = 0;
}
```

Scoping a variable inside a class (regardless of whether using `var`, `let` or `const`) will result in the following error: `Parsing error: Unexpected token`. It's not a very helpful error, which is why we mention it here. Students coming from other languages (such as C# or Ruby) may expect that JS will also have class variables, but that is not the case.

Instead, variables should always be scoped to methods inside the class (including the constructor). For instance, this is fine:

```
class Triangle {

  constructor() {
    this.variableScopedToConstructor = 0;
  }

  checkType() {
    let variableScopedToMethod = 0;
  }
}
```

The people behind ES6 made a conscious choice not to include class variables. The reasons for this are beyond the scope of this lesson; for now, it's enough to say that class variables simply don't fit JavaScript's prototypal inheritance model. In any case, variables should be scoped as tightly as possible as a best practice, so avoiding class variables (and global variables) is always a good idea.

While ES6's implementation of classes is mostly syntactic sugar, utilizing classes can make your code cleaner, more organized, and easier to read!

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/introduction-to-es6)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/es6-arrow-notation)

Lesson 36 of 49
Last updated more than 3 months ago.

disable dark mode

(http://www.epicodus.com)