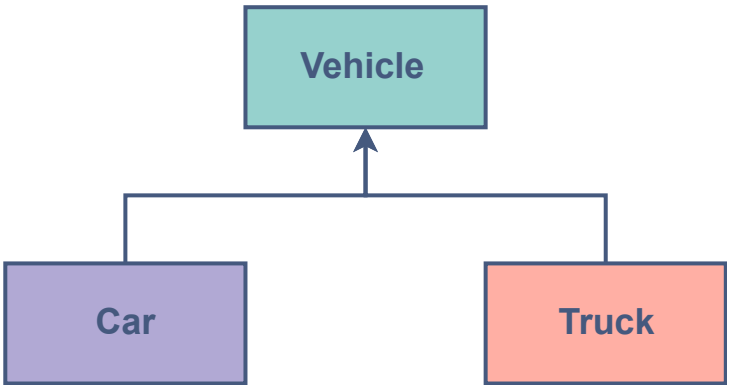
Class-based Inheritance in ES5

This lesson teaches us how class-based inheritance is implemented in the ES5 version by using constructor functions.

WE'LL COVER THE FOLLOWING

Inheritance Using Constructor Functions

In the last lesson, we learned how objects are inherited from the constructor function's prototype object using the prototype chain. The same concept could also be used in the implementation of class-based inheritance, i.e., where one class inherits the properties (including methods) from another class.



Classes Car and Truck inheriting from Vehicle Class

As discussed before, there was no `class` keyword in the ES5 version of JavaScript; hence we used constructor functions to implement a class. Now let's learn how class-based inheritance can be implemented using constructor functions, i.e., how one constructor function can inherit from another.

Example

Below is an example that is implementing class-based inheritance using constructor functions:

```
//constructor function Shape
function Shape(shapeName,shapeSides){
  this.name = shapeName
  this.sides = shapeSides
}
//defining the property "equalSides" on the prototype of Shape
Shape.prototype.equalSides = 'yes'
//displaying Shape.prototype
//it will show "equalSides" that is defined on the prototype
console.log(Shape.prototype)

//constructor function Rectangle
function Rectangle(shapeName,shapeSides,length,width){
  //call function invoked to inherit and initialize the properites of Shape for Rectangle
  Shape.call(this,shapeName,shapeSides)
  //properties of rectangle
  this.length = length
  this.width = width
}

//Setting Shape object to be the prototype of Rectangle
//so Rectangle can inherit Shape prototype properties
//through the new object created
Rectangle.prototype = Object.create(Shape.prototype)
//setting Rectangle's prototype constructor so that it points to itself
Rectangle.prototype.constructor = Rectangle
//defining a method "area" on the prototype of Rectangle
Rectangle.prototype.getArea = function(){
  return this.length*this.width
}
//displaying Rectangle.prototype
//it will show getArea that is defined on its prototype
//and constructor function pointing to Rectangle itself
console.log(Rectangle.prototype)
//creating a Rectangle object instance
var obj = new Rectangle('Rectangle',4,3,5)

//displaying properties
console.log("Name:",obj.name)
console.log("Sides:",obj.sides)
console.log("All sides are equal:",obj.equalSides)
console.log("Length:",obj.length)
console.log("Width",obj.width)
console.log("Area:",obj.getArea())
```

`call` Method

Before we get into the details of the code above, let's discuss what a `call`

Before we get into the details of the code above, let's discuss what a `call` function does.

The purpose of `call` is to invoke a method, defined anywhere, in the current context. Hence, `this` is passed as its first parameter. In the code above, `call` invokes the constructor method of the `Shape` class for `Rectangle`. It is passed the following parameters:

- `this` to specify the context, hence initializing the `Shape` constructor for `Rectangle`.
- `shapeName` and `shapeSides` are passed as parameters of the `Shape` constructor. However, these values will now get initialized for the `Rectangle` class.

Now, the `Rectangle` class will have the `Shape` class properties as well as its own.

Explanation

As seen from the code above:

- In **line 2**, a `Shape` function constructor is defined containing the *properties* `name` and `sides`.
- In **line 7**, the property `equalSides` is defined on `Shape.prototype`. Defining it on the prototype allows it to be inherited through prototypal inheritance.
- In **line 13**, another constructor function `Rectangle` is defined containing the *properties* `name`, `sides`, `length` and `width`.
- Inside the `Rectangle` in **line 15**, `Shape.call` initializes the properties present in `Shape` for `Rectangle`.
- In **line 23**, a `Shape` object is set as `Rectangle`'s prototype. It is created using `Object.create()` with `Shape.prototype` passed as its parameter.
    - `Object.create(Shape.prototype)` is equivalent to using `new Shape('Rectangle',4)` to create a new object. The only difference is that the object created using the former method won't have `name` and `sides` initialized. However, it will have access to the `Shape` prototype property which is the important part.

prototype property which is the important part.

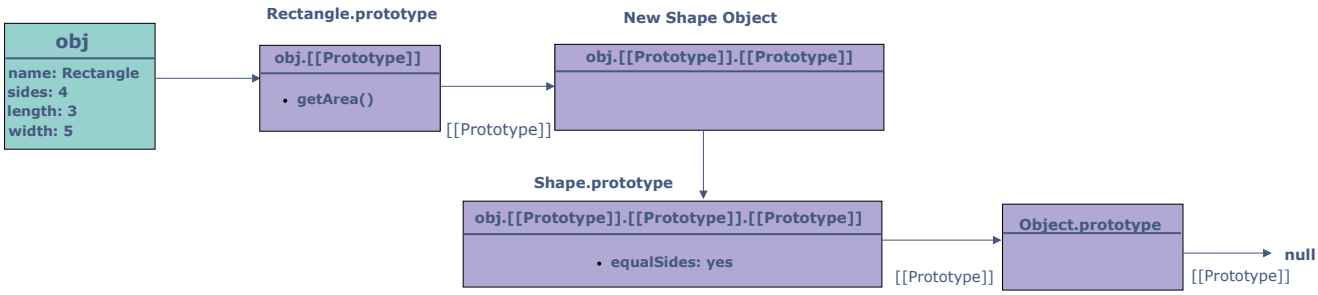
    - So, `Rectangle.prototype` will point to this new object of `Shape` class, whose `[[Prototype]]` property will point to `Shape.prototype`. Due to this, the `Rectangle` class will be able to inherit the `Shape` class properties that are defined on the prototype.

- Since `Rectangle.prototype` references to the `Shape` object, its constructor also points to `Shape()`. As discussed previously, it should point back to itself, hence, in **line 25** `Rectangle.prototype.constructor` is set equal to `Rectangle`.

- In **line 27**, the method `getArea` is defined on `Rectangle.prototype`. The function returns the *area* of a rectangle by computing the product of its `length` and `width`.

- In **line 35**, a new object instance, `obj`, is created from the *constructor function* `Rectangle`.

- This is what the prototype chain looks like at the end:



Prototype Chain

Accessing Properties & Methods

In the code above:

- Properties `name`, `sides`, `length` and `width` are all present in `obj`, hence they are directly accessed from there.

- When the method `getArea` is accessed:

    - First, `obj` is traversed.
    - It's not found in `obj`.
    - Next, `obj.[[Prototype]]` is traversed.
    - `obj.[[Prototype]]` contains `getArea` so the method is taken from there
- When `equalSides` is accessed:
    - First, `obj` is traversed.
        - It is not found in `obj`.
    - Next, `obj.[[Prototype]]` is traversed.
        - It is not found there either as it only contains `getArea`.
    - Next, `obj.[[Prototype]].[[Prototype]]` is traversed.
        - It is also not found there as the new `Shape` object that was set as `Rectangle`'s prototype doesn't contain `equalSides`.
    - Next, `obj.[[Prototype]].[[Prototype]].[[Prototype]]` is traversed.
        - It points to the `Shape` object's prototype which contains `equalSides`. Hence, it is inherited from there.

Now that you know how inheritance is implemented in the ES5 version, let's learn how to implement it in the ES6 version in the next lesson.