Prototypal Inheritance in ES5

introduction to prototypal inheritance using an example (rectangle "is a" shape)

Let's start with an example, where we implement a classical inheritance scenario in JavaScript.

```
function Shape( color ) {
    this.color = color;
}

Shape.prototype.getColor = function() {
    return this.color;
}

function Rectangle( color, width, height ) {
    Shape.call( this, color );
    this.width = width;
    this.height = height;
};

Rectangle.prototype = Object.create( Shape.prototype );
Rectangle.prototype.constructor = Rectangle;

Rectangle.prototype.getArea = function() {
    return this.width * this.height;
};

let rectangle = new Rectangle( 'red', 5, 8 );
console.log( rectangle.getArea() );
console.log( rectangle.getColor() );
console.log( rectangle.toString() );
```

`Rectangle` is a constructor function. Even though there were no classes in ES5, many people called constructor functions and their prototype extensions classes.

We instantiate a class with the `new` keyword, creating an object out of it. In ES5 terminology, constructor functions return new objects, having defined of properties and operations.

Prototypal inheritance is defined between `Shape` and `Rectangle`, as a *rectangle is a shape*. Therefore, we can call the `getColor` method on a rectangle, even though it is defined for shapes.

Prototypal inheritance is implicitly defined between `Object` and `Shape`. As the prototype chain is transitive, we can call the `toString` built-in method on a rectangle object, even though it comes from the prototype of `Object`.

If this example is your first encounter with classes, you may conclude that it is not too intuitive to read this example. Especially the two lines building the prototype chain seem far too complex to write. The definition of classes are also separated: we define the constructor and other methods in different places.

Now, let's talk about classes in ES6.