Under the Hood: Objects and Prototypes

Continuing the discussion on classes, this lesson will explain how JavaScript classes are different from classes we create in any other programming languages.

We'll cover the following

If you come from another programming background, chances are you already encountered classes and feel familiar with them. But as you'll soon discover, JavaScript classes are not quite like their C++, Java or C# counterparts.

JavaScript's Object-Oriented Model

To create relationships between objects, JavaScript uses *prototypes*.

In addition to its own particular properties, any JavaScript object has an internal property which is a link (known as a *reference*) to another object called its *prototype*. When trying to access a property that does not exist in an object, JavaScript tries to find this property in the prototype of this object. Here's an example (borrowed from Kyle Simpson's great book series You Don't Know JS).

```
1  const anObject = {
2    myProp: 2
3  };
4
5  // Create anotherObject using a
6  const anotherObject = Object.cr
7
8  console.log(anotherObject.myPro
```

In this example, the JavaScript statement `Object.create()` is used to create the object `anotherObject` with object `anObject` as its prototype.

```
// Create an object linked to myPrototypeObject
const myObject = Object.create(myPrototypeObject);
```

When the statement `anotherObject.myProp` is run, the `myProp` property of `anObject` is used since `myProp` doesn't exist in `anotherObject`.

If the prototype of an object does not have a desired property, then the search continues in the object's own prototype until we get to the end of the *prototype chain*. If the end of this chain is reached without having found the property, an attempted access to the property returns the value `undefined`.

```
const anObject = {
  myProp: 2
};

// Create anotherObject using anObject as a prototype
const anotherObject = Object.create(anObject);

// Create yetAnotherObject using anotherObject as a prototype
const yetAnotherObject = Object.create(anotherObject);

// myProp is found in yetAnotherObject's prototype chain (in anObject)
console.log(yetAnotherObject.myProp); // 2

// myOtherProp can't be found in yetAnotherObject's prototype chain
console.log(yetAnotherObject.myOtherProp); // undefined
```

This type of relationship between JavaScript objects is called *delegation*: an object delegates part of its operation to its prototype.

The True Nature of JavaScript Classes

In class-based object-oriented languages like C++, Java and C#, classes are static *blueprints* (templates). When an object is created, the methods and properties of the class are copied into a new entity, called an *instance*. After

properties of the class are copied into a new entity, called an *instance*. After

instantiation, the newly created object has no relation whatsoever with its class.

JavaScript's object-oriented model is based on prototypes, not classes, to share properties and delegate behavior between objects. In JavaScript, a class is itself an object, not a static blueprint. "Instantiating" a class creates a new object linked to a prototype object. Regarding classes behavior, the JavaScript language is quite different from C++, Java or C#, but close to other object-oriented languages like Python, Ruby and Smalltalk.

The JavaScript `class` syntax is merely a more convenient way to create relationships between objects through prototypes. Classes were introduced to emulate the class-based OOP model on top of JavaScript's own prototype-based model. It's an example of what programmers call syntactic sugar. The usefulness of the `class` syntax is a pretty heated debate in the JavaScript community.