Chapter 27: Inheritance

Section 27.1: Standard function prototype

Start by defining a `Foo` function that we'll use as a constructor.

```
function Foo (){}
```

By editing `Foo.prototype`, we can define properties and methods that will be shared by all instances of `Foo`.

```
Foo.prototype.bar = function() {
  return 'I am bar';
};
```

We can then create an instance using the `new` keyword, and call the method.

```
var foo = new Foo();

console.log(foo.bar()); // logs `I am bar`
```

Section 27.2: Difference between Object.key and Object.prototype.key

Unlike in languages like Python, static properties of the constructor function are *not* inherited to instances. Instances only inherit from their prototype, which inherits from the parent type's prototype. Static properties are never inherited.

```
function Foo() {};
Foo.style = 'bold';

var foo = new Foo();

console.log(Foo.style); // 'bold'
console.log(foo.style); // undefined

Foo.prototype.style = 'italic';

console.log(Foo.style); // 'bold'
console.log(foo.style); // 'italic'
```

Section 27.3: Prototypal inheritance

Suppose we have a plain object called `prototype`:

```
var prototype = { foo: 'foo', bar: function () { return this.foo; } };
```

Now we want another object called `obj` that inherits from `prototype`, which is the same as saying that `prototype` is the prototype of `obj`

```
var obj = Object.create(prototype);
```

Now all the properties and methods from `prototype` will be available to `obj`

```
console.log(obj.foo);
```

```
console.log(obj.bar());
```

Console output

```
"foo"
"foo"
```

Prototypal inheritance is made through object references internally and objects are completely mutable. This means any change you make on a prototype will immediately affect every other object that prototype is prototype of.

```
prototype.foo = "bar";
console.log(obj.foo);
```

Console output

```
"bar"
```

`Object.prototype` is the prototype of every object, so it's strongly recommended you don't mess with it, especially if you use any third party library, but we can play with it a little bit.

```
Object.prototype.breakingLibraries = 'foo';
console.log(obj.breakingLibraries);
console.log(prototype.breakingLibraries);
```

Console output

```
"foo"
"foo"
```

Fun fact I've used the browser console to make these examples and broken this page by adding that `breakingLibraries` property.

Section 27.4: Pseudo-classical inheritance

It's an emulation of classical inheritance using prototypical inheritance which shows how powerful prototypes are. It was made to make the language more attractive to programmers coming from other languages.

`Version < 6`

IMPORTANT NOTE: Since ES6 it doesn't make sense to use pseudo-classical inheritance since the language simulates conventional classes. If you're not using ES6, you should. If you still want to use the classical inheritance pattern and you're in a ECMAScript 5 or lower environment, then pseudo-classical is your best bet.

A "class" is just a function that is made to be called with the `new` operand and it's used as a constructor.

```
function Foo(id, name) {
    this.id = id;
    this.name = name;
}

var foo = new Foo(1, 'foo');
```

```
console.log(foo.id);
```

Console output

> 1

foo is an instance of Foo. The JavaScript coding convention says if a function begins with a capital letter case it can be called as a constructor (with the `new` operand).

To add properties or methods to the "class" you have to add them to its prototype, which can be found in the `prototype` property of the constructor.

```
Foo.prototype.bar = 'bar';
console.log(foo.bar);
```

Console output

> bar

In fact what Foo is doing as a "constructor" is just creating objects with `Foo.prototype` as it's prototype.

You can find a reference to its constructor on every object

```
console.log(foo.constructor);
```

> function Foo(id, name) { ...

```
console.log({ }.constructor);
```

> function Object() { [native code] }

And also check if an object is an instance of a given class with the `instanceof` operator

```
console.log(foo instanceof Foo);
```

> true

```
console.log(foo instanceof Object);
```

> true

Section 27.5: Setting an Object's prototype

`Version ≥ 5`

With ES5+, the `Object.create` function can be used to create an Object with any other Object as it's prototype.

```
const anyObj = {
    hello() {
        console.log(`this.foo is ${this.foo}`);
    },
};

let objWithProto = Object.create(anyObj);
objWithProto.foo = 'bar';

objWithProto.hello(); // "this.foo is bar"
```

To explicitly create an Object without a prototype, use `null` as the prototype. This means the Object will not inherit from `Object.prototype` either and is useful for Objects used for existence checking dictionaries, e.g.

```
let objInheritingObject = {};
let objInheritingNull = Object.create(null);

'toString' in objInheritingObject; // true
'toString' in objInheritingNull ; // false
```

Version ≥ 6

From ES6, the prototype of an existing Object can be changed using `Object.setPrototypeOf`, for example

```
let obj = Object.create({foo: 'foo'});
obj = Object.setPrototypeOf(obj, {bar: 'bar'});

obj.foo; // undefined
obj.bar; // "bar"
```

This can be done almost anywhere, including on a `this` object or in a constructor.

Note: This process is very slow in current browsers and should be used sparingly, try to create the Object with the desired prototype instead.

Version < 5

Before ES5, the only way to create an Object with a manually defined prototype was to construct it with `new`, for example

```
var proto = {fizz: 'buzz'};

function ConstructMyObj() {}
ConstructMyObj.prototype = proto;

var objWithProto = new ConstructMyObj();
objWithProto.fizz; // "buzz"
```

This behaviour is close enough to `Object.create` that it is possible to write a polyfill.