Advanced JavaScript

ITM - SDM S9

EPITA

Prototype

All Objects have an internal property denoted `[[Prototype]]`. It s simply a reference to another object. It is non-null at time of creation.

What's is the `[[Prototype]]` reference for? Operation `[[Get]]` is invoked when we reference a property on an object.

First time, `[[Get]]` checks if object itself has property on it and if yes, it's used

But if not, [[Get]] operation look up in `[[Prototype]]` link of the object

```
var anotherObject = {
    a: 2
};

// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

`Object.create(...)` creates an object with the `[[Prototype]]` linkage

But if `property` weren't found on `anotherObject`, the `[[Prototype]]` chain is again consulted and followed

If no matching property is ever found by the end of the chain, [[Get]] return `undefined`

This process continues until either a matching property name is found, or the `[[Prototype]]` chain ends.

If no matching property is ever found by the end of chain, the return result from the `[[Get]]` operation is `undefined`.

Where does the `[[Prototype]]` chain finish ?

`[[Prototype]]` chain finishes by the built-in `Object.prototype`.

This object includes common utilities like used by all Javascript objects like `.toString()`, `.valueOf()` and more.

Settings Shadowing properties

```
myObject.foo = 'bar';
```

If `myObject` has a normal data accessor property called `foo` present on it, the assignment is as simple as changing the value of the existing property.

If `foo` is not already present on `myObject`, the `[[Prototype]]` chain is traversed.

If `foo` is not found anywhere in the chain, the property `foo` is added directly to `myObject` with the specified value, as expected

If `foo` is present somewhere higher is the chain, a nuanced behavior can occur with the `myObject.foo = "bar"`

If property `foo` ends up both on `myObject` itself and at a higher level of the `[[Prototype]]` chain that starts at `myObject`, this called `shadowing`.

The `foo` property directly on `myObject` *shadows* any `foo` property which appears higher in the chain because the `myObject.foo` look-up would always find the `foo` property that's lowest in the chain.

Let's examine three scenario when `foo` is **not** already on `myObject` directly, but is at a higher level of `myObject`'s `[[Prototype]]` chain:

1. If data accessor property named `foo` is found anywhere higher on the `[[Prototype]]` chain, **and it's not marked as read-only then a new property called `foo` is added directly to `myObject`, resulting in** shadowed property.

2. If a `foo` found higher on the `[[Prototype]]` chain, but it's marked as ***read-only*** the setting and creation of the shadowed property on `myObject` ***are disallowed***

3. If a foo is found higher on the `[[Prototype]]` chain and it's a setter, then the setter will always be called. No `foo` will be added to myObject, nor will the `foo` setter be redefined.

You should try to avoid it if possible

Shadowing can even occur implicitly in subtle ways, so care must be taken if trying to avoid it. Consider:

```
var anotherObject = {
    a: 2
};

var myObject = Object.create( anotherObject );

anotherObject.a; // 2
myObject.a; // 2

anotherObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "a" ); // false

myObject.a++; // oops, implicit shadowing!

anotherObject.a; // 2
myObject.a; // 3

myObject.hasOwnProperty( "a" ); // true
```

"Class"

Javascript has just objects. In Javascript classes can't describe what an object can do. The object defines its own behavior directly.

Class "Functions"

All function by default get a public, non-enumerable property on them called `prototype`, which points at an otherwise arbitrary object.

```
function Foo() {
    // ...
}

Foo.prototype; // { }
```

Each object created from calling with `new` keyword in front of it will end up by `[[Prototype]]`.

```
function Foo() {
    // ...
}

var a = new Foo();

Object.getPrototypeOf( a ) === Foo.prototype; // true
```

When `a` is created by calling `new Foo()`, one the things that happens is that a gets an internal `[[Prototype]]` link to the object that `Foo.prototype` is pointing at.

In class-oriented languages, multiple copies (aka, "instances") of a class can be made.

This happens because the process of instantiating (or inheriting from) a class means, "copy the behavior plan from that class into a physical object", and this is done again for each new instance.

But in JavaScript, there are no such copy-actions performed. You don't create multiple instances of a class. You can create multiple objects that `[[Prototype]]` link to a common object. But by default, no copying occurs, and thus these objects don't end up totally separate and disconnected from each other, but rather, quite ***linked***.

`new Foo()` results in a new object (we called it `a`), and **that** new object `a` is internally `[[Prototype]]` linked to the `Foo.prototype` object.

We end up with two objects, linked to each other. That's it. We didn't instantiate a class. We certainly didn't do any copying of behavior from a "class" into a concrete object.

We just caused two objects to be linked to each other.

We can produce the same behavior with `Object.create(..)`

Constructors

```
function Foo() {
    // ...
}

Foo.prototype.constructor === Foo; // true

var a = new Foo();
a.constructor === Foo; // true
```

The `Foo.prototype` object by default gets a public, non-enumerable property called `.constructor` and this property is a reference back to the function that the object is associated with.

We see that object `a` created by "constructor" call `new Foo()` has a property called `.constructor` which points to "the function which created it".

"constructor" does not actually mean "was constructed by".

Functions aren't constructors, but function calls are "constructor calls" if and only if `new` is used.

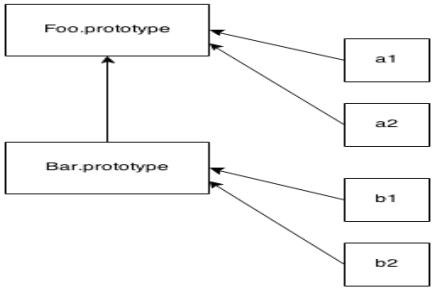
```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // create a new prototype object

var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

Prototypal (inheritance)

Delegation de `a1` to object `Foo.prototype` and `Bar.prototype` to `Foo.prototype`



This picture represent la delegation from `a1` to object `Foo.prototype`, and from `Bar.prototype` to `Foo.prototype`. which somewhat resembles the conceopt of Parent-Child class inheritance.

But the arrows represent the delegation rather than copy operations.

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

Foo.prototype.myName = function() {
    return this.name;
};

function Bar(name,label) {
    Foo.call( this, name );
    this.label = label;
}

Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.myLabel = function() {
    return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"
```

`Bar.prototype = Object.create(Foo.prototype)` says "make a new `Bar dot prototype` object that's linked" to `Foo dot prototype`

We will have realise this work with other was but they does not work as we expect:

```
Bar.prototype = Foo.prototype
```

This statement doesn't create a new object for `Bar.prototype` It just creates `Bar.prototype` be another reference to `Foo.prototype`, which links `Bar` directly to ***the same object as*** as `Foo`.

This means when you start assigning, like `Bar.prototype.myLabel = ...` you're modifying ***not a separate object*** but the **shared** `Foo.prototype`.

```
Bar.prototype = new Foo();
```

Create link to `Foo.prototype` as we'd want. But it uses the `Foo(...)` "constructor call" to do it. If that function has any side-effects, those side-effects happen at the time of this linking, rather than only when the eventual `Bar()` "descendants" are created, as would likely be expected.

So, we use `Object.create(..)` to make a new object that's properly linked. The slight downside is that we have to create a new object, throwing the old one away, instead of modify the existing defaut object we provided.

In ES6 version and more, there is une method which called : `Object.setPrototypeOf(..)` that modify in reliable way the linkage of an existing object.

```
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

Instropection

What do you do if you have an object and want to find out? What object it delegates to?

Inspection of instance for inheritance ancestry id often called **instrospection**

```
function Foo() {
    // ...
}

Foo.prototype.blah = ...;

var a = new Foo();
```

```
a instanceof Foo;
```

We can do this with the `instanceof` operator.

`instanceof` operator takes object as its left-hand operand and a ***function*** as its right-hand operand.

`instanceof` operator answers the following question:

in the entire `[[Prototype]]` chain of a, does the object arbitrarily pointed to by Foo.prototype ever appear?

Operator `instanceof` can only inquire about the object's ancestry if we have it with its attached prototype reference.

So instance operator cannot help us if we want to know if two object are related to each other through the `[[Prototype]]` chain.

The second approach to realise introspection is : `IsPrototypeOf`.

Here we just need an object to test against an another object.

`IsPrototypeOf`answer the following question:

in the entire `[[Prototype]]` chain of a, does Foo.prototype ever appear?

```
    var foo = {};

    var bar = Object.create(foo)
    //does `bar` appera anywhere in
    // `foo` `[[Prototype]]` chain?
    bar.isPrototypeOf(foo) // true
```

We have method that allow to retrieve `[[Prototype]]` :
`Object.getPrototypeOf(a)`

Most web Browsers have that allow to retrieve prototype : .**proto**

It is very helpful for inspect prototype chain **proto** is a non-enumerable property of Object.prototype.

We could implemented *_proto* like this:

```
Object.defineProperty( Object.prototype, "__proto__", {
    get: function() {
        return Object.getPrototypeOf( this );
    },
    set: function(o) {
        // setPrototypeOf(..) as of ES6
        Object.setPrototypeOf( this, o );
        return o;
    }
} );
```

Object Links

The `[[Prototype]]` mechanism is an internal link that existing on one object which references some other object.

```
var foo = {
    something: function() {
        console.log( "Tell me something good..." );
    }
};

var bar = Object.create( foo );

bar.something(); // Tell me something good...
```

`Object.create(...)` creates a new object linked to the object we specified, which give us all the power of the prototype mechanism.

Note, that `Object.create(null)` creates an that has an empty `[[Prototype]]` linkage, so the object cannot delegate anywhere.

This object are often called "dictionnaries" as they are used purely for storing data in properties.

Linking as fallback

It will be tempting to think that `[[Prototype]]` is a sort fallback for missing properties or methods.

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.cool(); // "cool!"
```

This type code work by virtue of `[[Prototype]]`, it seems magical but is harder to understand and maintain.

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.doCool = function() {
    this.cool(); // internal delegation!
};

myObject.doCool(); // "cool!"
```

We call `myObject.doCool()`, which is a method that actually exist on `myObject`, making our API design more explicit.

This implementation follows **the delegation design pattern**.