Advanced JavaScript

ITM - SDM S9

EPITA

Delegation

Comparison between Delegation and Class

We going to see the difference between this patterns through an example which represents some similar tasks ("XYZ", "ABC").

Class theory

With classes to realize this, we define a parent class like `Task`, defining the common behavior.

Then, we define child class `ABC` and `XYZ`. Both, we inherited from `Task` and each of which specialized behavior to handle their respective tasks.

```
class Task {
    id;

    Task(ID) { id = ID; }
    outputTask() { output( id ); }
}

class XYZ inherits Task {
    label;

    XYZ(ID,Label) { super( ID ); label = Label; }
    outputTask() { super(); output( label ); }
}

class ABC inherits Task {
    // ...
}
```

Delegation

Let's try to realize the same behavior with a delegation.

```
var Task = {
    setID: function(ID) { this.id = ID; },
    outputID: function() { console.log( this.id ); }
};

// make `XYZ` delegate to `Task`
var XYZ = Object.create( Task );

XYZ.prepareTask = function(ID,Label) {
    this.setID( ID );
    this.label = Label;
};

XYZ.outputTaskDetails = function() {
    this.outputID();
    console.log( this.label );
};

// ABC = Object.create( Task );
// ABC ... = ...
```

First, we create an object called `Task`, it will have the utility method that various tasks can use.

Then, we define two specifics tasks `(XYZ, ABC)` for each task we create an object to hold that task-specific data/behavior and we link specifics tasks to `Task` utility object.

In this code, there are not classes or functions only objects.

`XYZ` is set up via `Object.create(..)` to `[[Prototype]]` delegate to the `Task` object.

As compared to class orientation(OO) this style of code call OLOO(objects-linked-to-other-objects)

In Javascript, the `[[Prototype]]` mechanism links **objects** to other **objects**. There are not mechanisms like `classes`

Some differences to note with **OLOO style code:**

1. With class design pattern, we intentionally named `method` `outputTask`, the same on both parent (Task) and child(XYZ), so that we could take advantage of overriding (polymorphism). In behavior delegation, **we avoid if at all possible naming things the same** at different levels of the [[Prototype]] chain, because we want to avoid the name collisions.

   The design pattern calls for less of general method names which are prone to overriding and instead more of descriptive method names, specific to the type of behavior each object is doing.

2. `this.setID(ID);` inside of a method on the `XYZ`, object first looks on `XYZ` for `setID(...)`, but since it does not find a method of that name on XYZ, [[Prototype]] delegation, means it can follow the link to `Task` to look for `setID(...)`, which it of course finds. Moreover, because of implicit call-site this binding rules when setID(..) runs, even though the method was found on Task he this binding for that function call is XYZ exactly as we'd expect and want. We see the same thing with this.outputID() later in the code listing.

   In other words, the general utility methods that exist on Task are available to us while interacting with XYZ, because XYZ can delegate to Task.

Behavior Delegation means: let some object (XYZ) provide a delegation (to Task) for property or method references if not found on the object (XYZ).

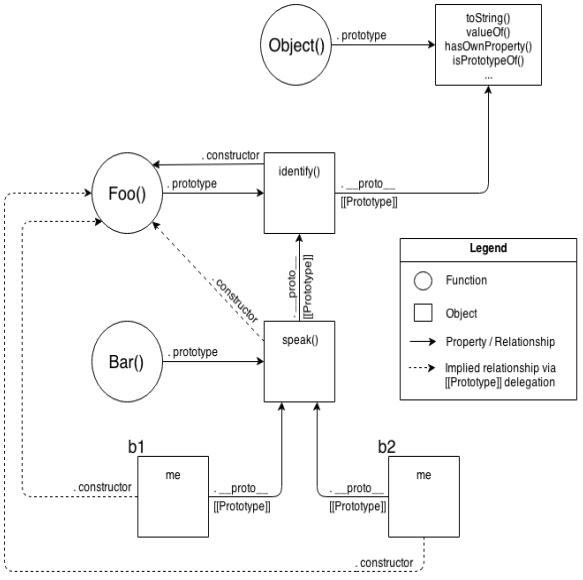
Delegation is an extremely *powerful* design pattern. With inheritance, the objects are organized in our mind vertically, Children are descended from Parents.

Think of objects side-by-side, as peers, with any direction of delegation links between the objects as necessary.

Mental Models Compared

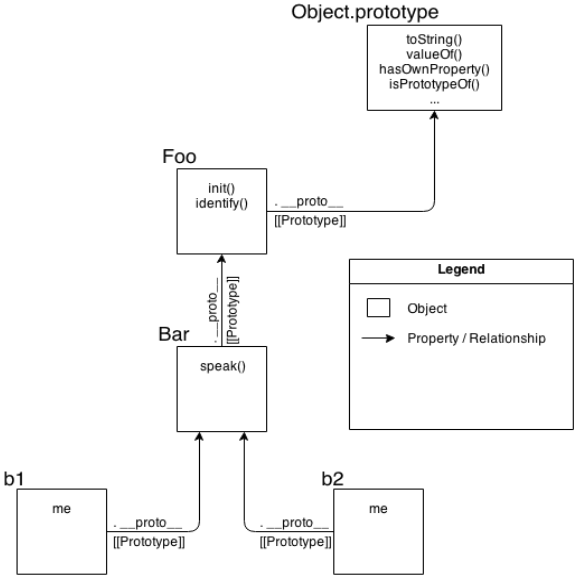
```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call( this, who );
}
Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

Parent class Foo, inherited by child class Bar, which is then instantiated twice as b1 and b2. What we have is b1 delegating to Bar.prototype which delegates to Foo.prototype. This should look fairly familiar to you, at this point. Nothing too ground-breaking going on.


Object()
. prototype
toString()
valueOf()
hasOwnProperty()
isPrototypeOf()
...
. constructor
Foo()
. prototype
identify()
. __proto__
[[Prototype]]
. constructor
. __proto__
[[Prototype]]
Bar()
. prototype
speak()
Legend
Function
Object
Property / Relationship
Implied relationship via
[[Prototype]] delegation
b1
me
. constructor
. __proto__
[[Prototype]]
b2
me
. __proto__
[[Prototype]]
. constructor

```
var Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};

var Bar = Object.create( Foo );

Bar.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = Object.create( Bar );
b1.init( "b1" );
var b2 = Object.create( Bar );
b2.init( "b2" );

b1.speak();
b2.speak();
```

We take exactly the same advantage of [[Prototype]] delegation from b1 to Bar to Foo as we did in the previous snippet between b1, Bar.prototype, and Foo.prototype. We still have the same 3 objects linked together.



The OLOO style code is less complicated than OO-style because OLOO-style code embraces the fact that the only thing we ever really cared about was the **objects linked to other objects**.

The class more complicated than delegation, we get the same end results, So, don't use any more class pattern in JS.

Classes vs. Objects

See the example that reveals the difference between Class and Delegation.

Let's see the difference through the creation of widgets (buttons, drop-downs, etc).

- With the OO design pattern:

```
// Parent class
function Widget(width,height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
}

Widget.prototype.render = function($where){
    if (this.$elem) {
        this.$elem.css( {
            width: this.width + "px",
            height: this.height + "px"
        } ).appendTo( $where );
    }
};
```

```
// Child class
function Button(width,height,label) {
    // "super" constructor call
    Widget.call( this, width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
}

// make `Button` "inherit" from `Widget`
Button.prototype = Object.create( Widget.prototype );

// override base "inherited" `render(..)`
Button.prototype.render = function($where) {
    // "super" call
    Widget.prototype.render.call( this, $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.prototype.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

OO design patterns tell us to declare a base `render(...)` in in the parent class then override it in our child class.

ES6 class sugar

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}
```

```
class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

The syntax more beautiful than the previous example. The presence of `super(...)` in particular seems nice though when you dig into it, it's not all roses!.

Despite syntactic improvements, **these are not real classes** , as they still operate on top of the [[Prototype]] mechanism

With the OLOO design pattern:

```
var Widget = {
    init: function(width,height){
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    },
    insert: function($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
};
```

```
var Button = Object.create( Widget );

Button.setup = function(width,height,label){
    // delegated call
    this.init( width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
};
Button.build = function($where) {
    // delegated call
    this.insert( $where );
    this.$elem.click( this.onClick.bind( this ) );
};
Button.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );

    var btn1 = Object.create( Button );
    btn1.setup( 125, 30, "Hello" );

    var btn2 = Object.create( Button );
    btn2.setup( 150, 40, "World" );

    btn1.build( $body );
    btn2.build( $body );
} );
```

In this OLOO style approach `Widget` is **just an object** and is sort of utility collection that any specific type of widget might want to delegate to. `Button` **is also just stand-alone object** .

We didn't share the same method name `render(...)` in both objects, but instead we chose different names (`insert(...)` and `build(..)`) that was more descriptive of what task each does specifically. With OLOO style we don't need `.prototype` or `new` because they are unnecessary.

Nicer Syntax

One of nicer things that makes ES6's `class` is the short-hand syntax for declaring class methods:

```
class Foo {
    methodName() { /* .. */ }
}
```

We can drop the word `function`.

As of ES6, we can use *concise method declarations* in any object literal, so an object in OLOO style can be declared this way:

```
var LoginController = {
    errors: [],
    getUser() { // Look ma, no `function`!
        // ...
    },
    getPassword() {
        // ...
    }
    // ...
};
```

In comparison with `class` syntax, this object literals will still require `,` comma between separators.

There is one drawback to concise methods that are subtle but important to note.

```
var Foo = {
    bar() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

Here's the syntactic de-sugaring that expresses how that code will operate:

```
var Foo = {
    bar: function() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

The difference is that the `bar()` short-hand became an *anonymous function expression* attached to the `bar` property, because the function object itself has no name identifier.

Use *anonymous function expression* involves three main downsides:

1. makes debugging stack traces harder
2. makes self-referencing (recursion, event (un)binding, etc) harder
3. makes code (a little bit) harder to understand

Introspection

Introspection with OLOO is extremely simplified because we have plains object that is related via [[Prototype]] delegation.

```
var Foo = { /* .. */ };

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

var b1 = Object.create( Bar );
```

```
// relating `Foo` and `Bar` to each other
Foo.isPrototypeOf( Bar ); // true
Object.getPrototypeOf( Bar ) === Foo; // true

// relating `b1` to both `Foo` and `Bar`
Foo.isPrototypeOf( b1 ); // true
Bar.isPrototypeOf( b1 ); // true
Object.getPrototypeOf( b1 ) === Bar; // true
```

We're not using `instanceof` anymore, now we just ask the question "are you prototype of me?". There's no more indirection necessary with stuff like Foo.prototype or the painfully verbose Foo.prototype.isPrototypeOf(..).