Spred and Rest Operator

ES6 introduces a new `...` operator that's typically referred to as the *spread* or *rest* operator, depending on where/how it's used.

```
function foo(x,y,z) {
    console.log( x, y, z );
}

foo( ...[1,2,3] );  // 1 2 3
```

But `...` can be used to spread expand a value in other contexts as well, such as inside another array declaration:

```
var a = [2,3,4];
var b = [ 1, ...a, 5 ];

console.log( b );                    // [1,2,3,4,5]
```

The other common usage of `...` can be seen as essentially the opposite; instead of spreading a value out, the `...` gathers a set of values together into an array. Consider:

```
function foo(x, y, ...z) {
    console.log( x, y, z );
}

foo( 1, 2, 3, 4, 5 );            // 1 2 [3,4,5]
```

The ...z in this snippet is essentially saying: "gather the rest of the arguments (if any) into an array called z."

THIS

To answer a question: what this `this` a reference to ? We must to determine the call-site: the place in code where a function is called

We must determine which of 4 rules applies I'm going to explain this 4 rules.

First rules: standalone function invocation

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo(); // 2
```

When `foo()` is called, `this.a` resolves to our global variable a because in this case the default binding for this applies to the function call, and so points this at the global object.

Standalone rules is apply when the call-site of function is located with a plain, un-decorated function reference

Implicit Binding

Second rule to consider is: does the call-site have context object

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

The call-site uses the `obj` context to reference the function. The call of `foo()` is preceded by an object reference to `obj`. When there is a context object for function reference. The implicit binding rule says that it's that object which should be used for the function call's this binding.

The last level of an object property reference chain matters to the call-site

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

Implicit lost

Sometimes,the implicity bound function loses this binding which usually means it falls back to the default binding, of either the global object or undefined, depending on strict mode.

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo;  // function reference/alias!

var a = "oops, global"; // `a` also property on global object

bar(); // "oops, global"
```

Even though `bar` appears to be a reference to `obj.foo`, in fact, it's really just another reference to `foo` itself. Moreover, the call-site is what matters, and the call-site is `bar()`, which is a plain, un-decorated call and thus the default binding applies.

Explicit Binding

What if you want to force function call to use a particular object for `this` binding, without putting a property function reference on a object ?

To realise this we can use to function `call(...)` and `apply(...)`. They both take, as their first parameter, an object to use for the `this`, and then invoke the function with that `this` specified

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Invoking `foo` with explicit binding by `foo.call(..)` allows us to force its this to be obj.

New binding

In Javascript `constructors` are just a functions that happpen to be called with the `new` in front of them. They are not attached to classes, nor are they instantiating a class. They are not even special types of functions. They're just regular functions that are, in essence, hijacked by the use of new in their invocation.

When a function is invoked with new in front of it, otherwise known as a constructor call, the following things are done automatically:

1. a brand new object is created (aka, constructed) out of thin air
2. the newly constructed object is [[Prototype]]-linked
3. the newly constructed object is set as the this binding for that function call
4. unless the function returns its own alternate object, the new-invoked function call will automatically return the newly constructed object.

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

By calling foo(..) with new in front of it, we've constructed a new object and set that new object as the this for the call of foo(..). So new is the final way that a function call's this can be bound. We'll call this **new binding**.

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".

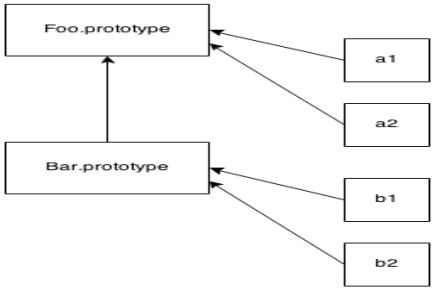
"constructor" does not actually mean "was constructed by".

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**.

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).

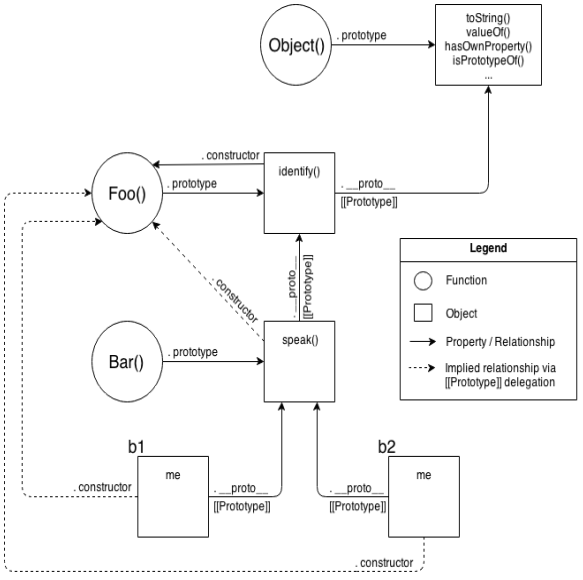
Delegation is an extremely *powerful* design pattern. With inheritance, the objects are organized in our mind vertically, Children are descended from Parents.

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

Mental Models Compared

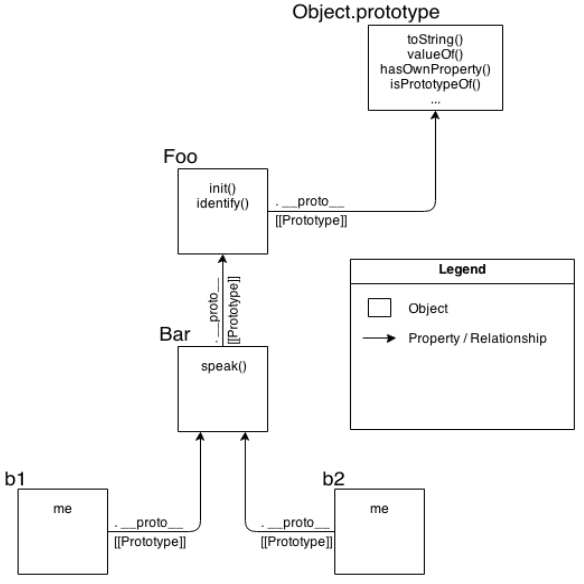
```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

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]]
Legend
Function
Object
Property / Relationship
Implied relationship via
[[Prototype]] delegation
. __proto__
[[Prototype]]
. constructor
Bar()
. prototype
speak()
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(..).