Class / Prototype in JS

Class / Inheritance descibres a certain form of code organization and architecture — a way of modeling real world problem domains in our software. (Also referred as **O**bject-**O**riented programming - OO in short)

Class as a design pattern covers mechanism as *instantiation*, *inheritance*, *polymorphism* and some more specific design patern like *Iterator*,*Obserever*, *factory* etc. Class are in fact one of several common design approach, as well as *functionnal programming* by example

Difference between traditionnal classes and JS classes

- When traditionnal classes are instanciated, a copy of behavior from class instance occurs. Same goes when class are inherited, a copy of the behaviors from parent to child occurs.
- Javascript class is similar in syntax but **does not** automatically create copies between objects

JS does not have classes in a strict sense. Javascript relies on its object mechanism to have a syntax close to the class one.

Plainly there are no classes in javascript to instantiate, only objects. And objects don't get copied to other objects they get linked together. Since class in others languages involve copies - Javascript developers have to use a different type of mechanism to mimic the copy : `mixins`

Relative Polymorphism

relative polymorphism is when a method reference an other method at a higher level if the inheritance hierarchy. It's called *relative* here as we do not specify which inheritance level we want but just look one lelvel up.

```
//pseudoCode

class Vehicle{
  drive(){
    output("starting engine")
  }
}

class Car inherits Vehicle{
  drive(){
    inherited:drive()
    output("rolling on 4 wheels")
  }
}
```

Javascript's object mechanism does not automatically perform copy behavior when you inherit or instantiate. As Object don't get copied to other object (they get link together), javascript community has come up with **mixin**

Mixin

```
// vastly simplified `mixin(..)` example:
function mixin( sourceObj, targetObj ) {
  for (var key in sourceObj) {
    // only copy if not already present
    if (!(key in targetObj)) {
      targetObj[key] = sourceObj[key];
    }
  }

  return targetObj;
}

var Vehicle = {
  engines: 1,

  ignition: function() {
    console.log( "Turning on my engine." );
  },
```

```
  drive: function() {
    this.ignition();
    console.log( "Steering and moving forward!" );
  }
};

var Car = mixin( Vehicle, {
  wheels: 4,

  drive: function() {
    Vehicle.drive.call( this );
    console.log( "Rolling on all " + this.wheels + " wheels!" );
  }
} );
```

`Car` now has a copy of the properties and function from `Vehicle` . Functions are not technically dupplicated but rather copy the reference to the functions itselves.

`Vehicle.drive.call( this );` refers to **explicit pseudopolymorphism**. Because both `Car` and `vehicule` have a function of the same name, to distinguish them we need to make an absolute reference.

If we have said `Vehicle.drive()` instead, the `this` binding for that function would have been the `Vehicle` Object instead (implicit binding rule)

As it creates manual/explicit linkage in every single function, the result of such approach is usually more complex, hard to maintain code.

[[Prototype]] Chain

Objects in Javascript have internal property, denoted in the specification as **[[Prototype]]** which is simply a reference to an other object.

The **[[Prototype]]** chain is used through the [[Get]] operation : if it cannot find the requested property on the object directly, the operation proceeds to follow the protoptype link :

```
const anotherObject = {
  a:2
}

//Create an object linked to anotherObject
const myObject = Object.create(anotherObject)

myObject.a; // 2
```

If no match property name is ever found at the end of the **[[Prototype]]** chain, the return result from the [[Get]] operation is undefined.

The top of every *normal* [[Prototype]] chain is the built in `Object.prototype`

Several scenarios with :

```
myObject.foo = "bar"
```

- if `myObject` has already a *normal data accessor* property called `foo` , then the new assignment simply changes its value
- if `myObject` doesn't have a `foo` property then the [[Prototype]] chain is traversed. If `foo` is not found at the end, the foo property is added directly on `myObject`
- if a *normal data accessor* property called `foo` is found higher on the chain (and not marked as read-only `writable:true`) the foo property is added directly on `myObject` resulting in a **shadowed** property
- if a *normal data accessor* property called `foo` is found higher on the chain (and **marked** as read-only `writable:false`) then no property is created on `myObject` as it's disallowed.

Usually shadowing is more complicated and nuanced than it's worth so you should try to avoid it.

```
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
```

Though it may appear that `myObject.a++` should (via delegation) look-up and just increment the `anotherObject.a` property itself *in place*, instead the `++` operation corresponds to `myObject.a = myObject.a + 1`

The result is `[[Get]]` looking up `a` property via `[[Prototype]]` to get the current value `2` from `anotherObject.a`, incrementing the value by one, then `[[Put]]` assigning the `3` value to a new shadowed property `a` on `myObject`. Oops!

Function.prototype

Each object created from calling a `function` with `new` keyword will end up [[Prototype]]-linked to this "function dot prototype" object.

```
function Foo() {}

const a = new Foo()

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

`new Foo()` results in a new object, and that new object, 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 just caused two objects to be linked together through a kind of accidental side effect of `new` Foo call.

How de we call that ?

This mechanism is often called *prototypal inheritance*. It's misleading as it tries to capitalize on `inheritance` mechanism from traditionnal class oriented world. But **inheritance** implies `copy` where Javascript is only about **linking** objects. **Delegation** is a much more appropriate term for Javascript object linkage mechachanism.

Constructors

It's tempting to think that `Foo` is a "constructor", because we call it with `new` and we observe that it "constructs" an object but there is no constructor in javascript. In other words, in JavaScript, it's most appropriate to say that a "constructor" is **any function called with the new keyword** in front of it.

To further the confusion of "constructor" semantics, the arbitrarily labeled `Foo.prototype` object has another trick up its sleeve. Consider this code:

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

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

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

The `Foo.prototype` object by default (at declaration time on line 1 of the snippet!) gets a public, non-enumerable property called `.constructor`, and this property is a reference back to the function `Foo` that the object is associated with. (since a function is an Object `sub-type`)

Moreover, we see that object `a` created by the "constructor" call `new Foo()` *seems* to also have a property on it called `.constructor` which similarly points to "the function which created it".

In the code snippet above constructor **does not mean** was constructed by. `a` has not any `.constructor` property on itself - it gets it from [[Prototype]] delegation from `Foo` . Furthermore `Foo.prototype.constructor` is only there by default on the object created when `Foo` the function is declared.

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

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

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

`a1` has no `.constructor` property, so it delegates up the `[[Prototype]]` chain to `Foo.prototype`.

But that object doesn't have a `.constructor` either as we changed the default, so it keeps delegating, this time up to `Object.prototype`, the top of the delegation chain. *That* object indeed has a `.constructor` on it, which points to the built-in `Object(..)` function.

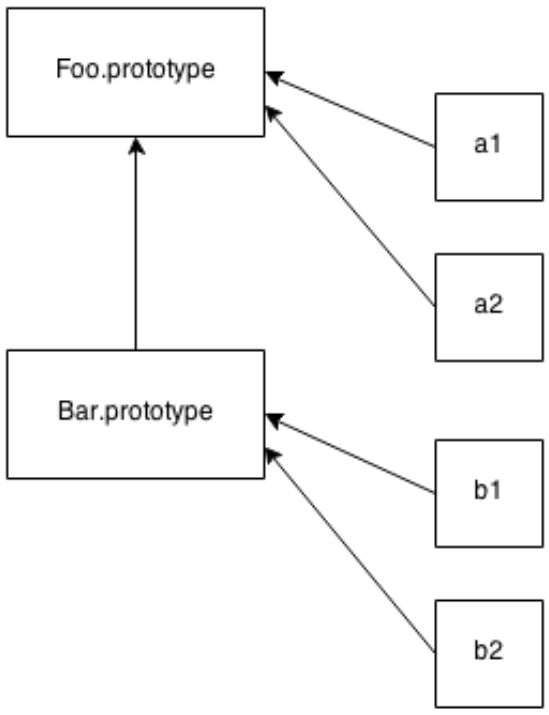
`.constructor` is extremely unreliable, and an unsafe reference to rely upon in your code. **Generally, such references should be avoided where possible.**

Prototypal Inheritance (or truly delegation)

We've already seen how an object can "inherit" from an other object. But Inheritance is traditionnaly the mechanism from which a class inherits from an other and not the relation between an instance and a class.

See the 2 relations below :

- b1/b2 to Bar.prototype
- Bar.prototype to Foo.prototype



`Bar.prototype` to `Foo.prototype`, which somewhat resembles the concept of Parent-Child class inheritance. *Resembles*, except of course for the direction of the arrows, which show these are delegation links rather than copy operations.

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

Foo.prototype.myName = function() {
  console.log('found you')
};

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

// here, we make a new `Bar.prototype`
// linked to `Foo.prototype`
Bar.prototype = Object.create( Foo.prototype );

// Beware! Now `Bar.prototype.constructor` is gone,
// and might need to be manually "fixed" if you're
// in the habit of relying on such properties!
```

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

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

a.myName(); // "a"
a.m
yLabel(); // "obj a"
```

Here we use `Object.create( Foo.prototype )` to create the delegation chain.

When `function Bar() { .. }` is declared, `Bar`, like any other function, has a `.prototype` link to its default object. But *that* object is not linked to `Foo.prototype` like we want. So, we create a *new* object that *is* linked as we want.

Below 2 general mistakes people do trying to create the same delegation link

```
//doesn't work
Bar.prototype = Foo.prototype

//work with side effect
Bar.prototype = new Foo()
```

`Bar.prototype = Foo.prototype` doesn't assign a new object but makes it a reference to `Foo.prototype`, therefore anychange on Bar.prototype would affect Foo.prototype

`Bar.prototype = new Foo()` creates a new object that is linked to Foo.prototype. But as it uses the Foo 'constructor' call any side effect of the function Foo will be called.

With ES6 we have the option to directly modify the prototype without throwing the old one.

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

Reflection and "class" relation inspection

Inspecting an instance (object in JS) for its inheritance ancestor (delegation linkage in JS) is called **reflection** in traditionnal language.

2 ways to do reflection in JS

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

Foo.prototype.add = ...;
const a = new Foo()

a instanceof Foo; //true

// isPrototyopeOf
Foo.prototype.isPrototypeOf(a)
```

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

But the big advantage of `isPrototypeOf()` is that it doesn't require a function, you could apply it directly on 2 objects

```
obect1.isPrototypeOf(object2) //works
```

There are 2 direct ways to retrieve the prototype from an object:

```
Object.getPrototypeOf(object1)

object1.__proto__
```

Note that `__proto__` doesn't exist on the object itself but on the built in `Object.prototype`

From : http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/

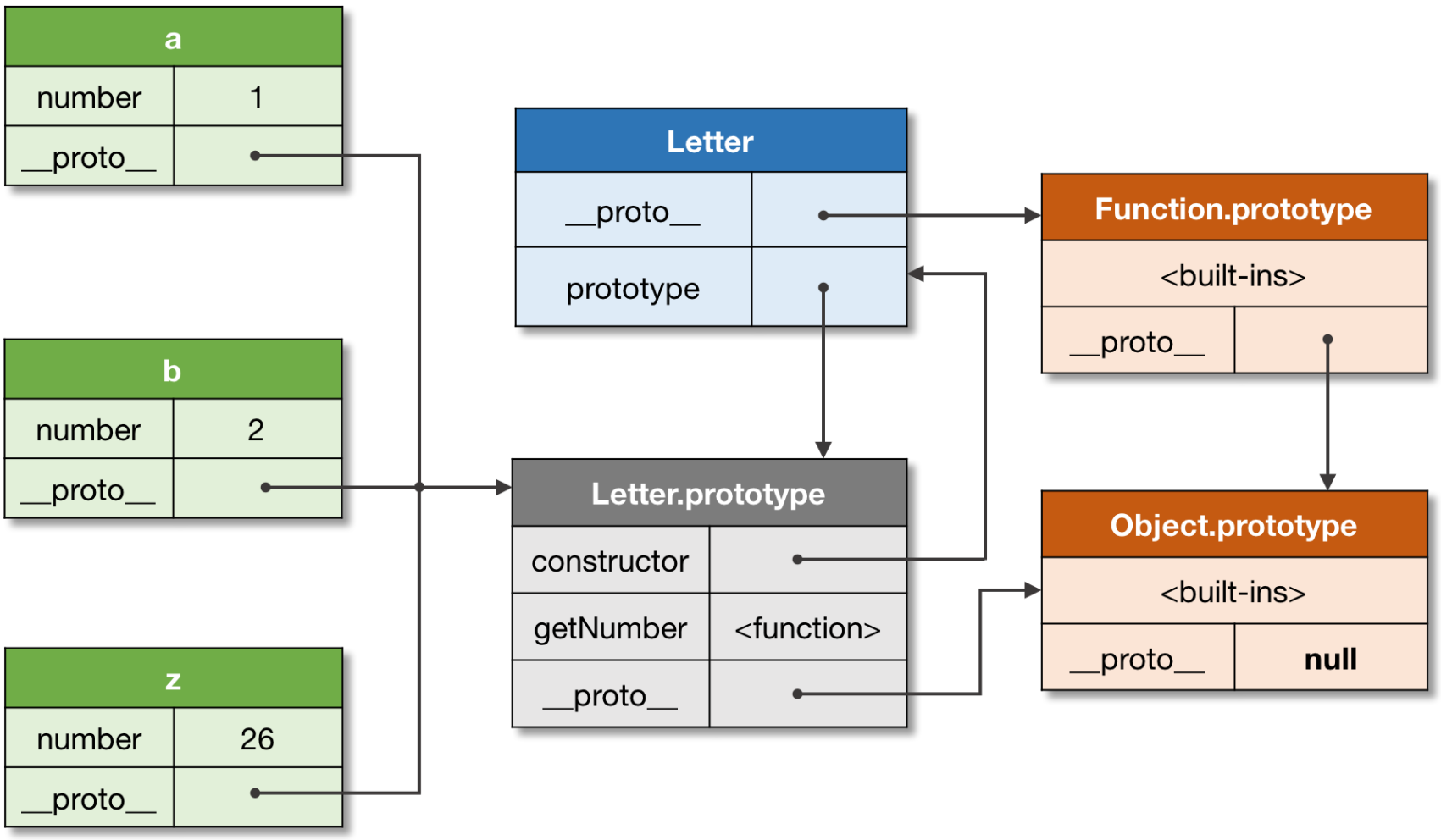
```
function Letter(number) {
  this.number = number;
```

```
}

Letter.prototype.getNumber = function() {
  return this.number;
};

let a = new Letter(1);
let b = new Letter(2);
// ...
let z = new Letter(26);

console.log(
  a.getNumber(), // 1
  b.getNumber(), // 2
  z.getNumber(), // 26
);
```



The figure above shows that *every object* has an associated prototype. Even the constructor function (class) `Letter` has its own prototype, which is `Function.prototype`. Notice, that `Letter.prototype` is the prototype of the Letter *instances*, that is `a`, `b`, and `z`.

The *actual* prototype of *any* object is *always* the `__proto__` reference. And the explicit `prototype` property on the constructor function is just a reference to the prototype of its *instances*; from instances it's still referred by the `__proto__`. See details here.

ES6 Classes

```
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!" );
  }
}
```

Main Benefits :

- there is no more reference to `prototype` confusing code :
  - `Button` is declared directly to "inherit from" aka `extends` instead of using `Object.Create()`
  - `super()` allow direct *relative polymorphism* so that any method can refer **one** level up in the chain to a method with the same name.