COMPS350F Project

Part II

1. Divide and Conquer (JIAO)

2. JavaScript Types

In this part of report, we will be applying the software engineering concepts to analysis the built-in types of JavaScript language.

Let's start off by looking at the types. JavaScript programs manipulate values, and those values all belong to a type. There are only two kinds of types – **Primitives** and **Objects**.

- **Primitives**
  - Number
  - BigInt
  - String
  - Boolean
  - Null
  - Undefined
- **Objects**
  - Function
  - Array
  - Date
  - RegExp

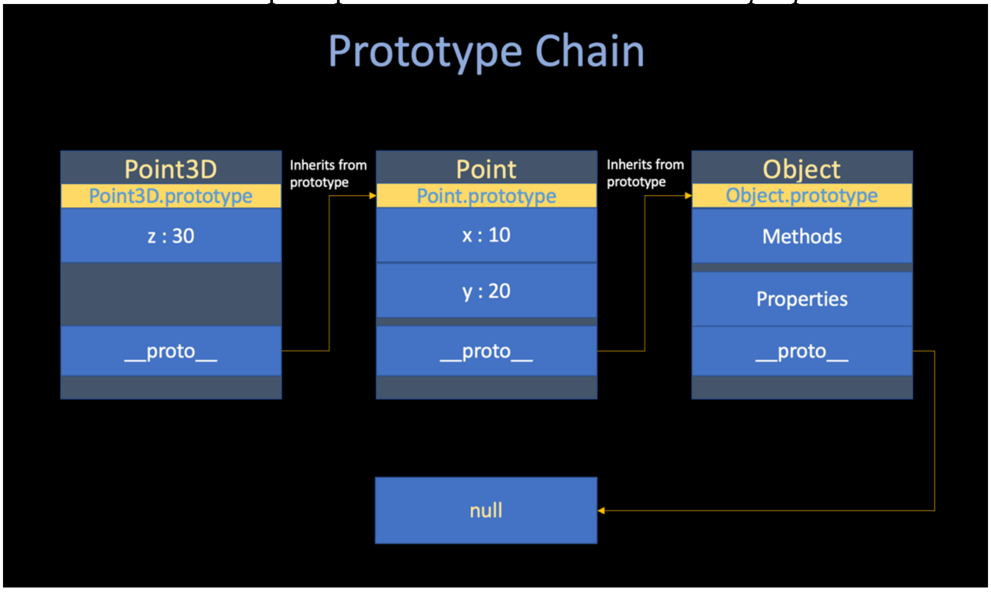
From the implementation-level perspective, **all-of these different types can be implemented with object.**

1. Object

An **object** is a collection of properties and has its own **prototype** – either another object or null. The prototype is a delegation object used to implement **prototype-based inheritance** and can be set explicitly via either the `__proto__` property or `Object.create` method.

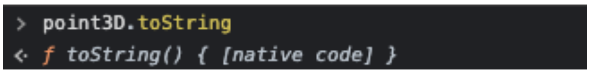
```
let point = { // point is an object
    x: 10,
    y: 20,
};

// either this way
let point3D = {
    z: 30,
    __proto__: point,
}


// or this way
let point3D = Object.create(point);
point3D.z = 30;

console.log(
    point3D.x, // 10
    point3D.y, // 20
    point3D.z, // 30
);
```

The **Prototype Chain**, which is a finite chain of object.
Shows how JavaScript implement *inheritance and shared properties.*



If the property is not found in the object itself, the rule is very simple:

- There is an attempt to resolve it in the prototype, the prototype of the prototype etc.
- After the whole prototype chain is considered, if a property eventually not found, the `undefined` value is returned.



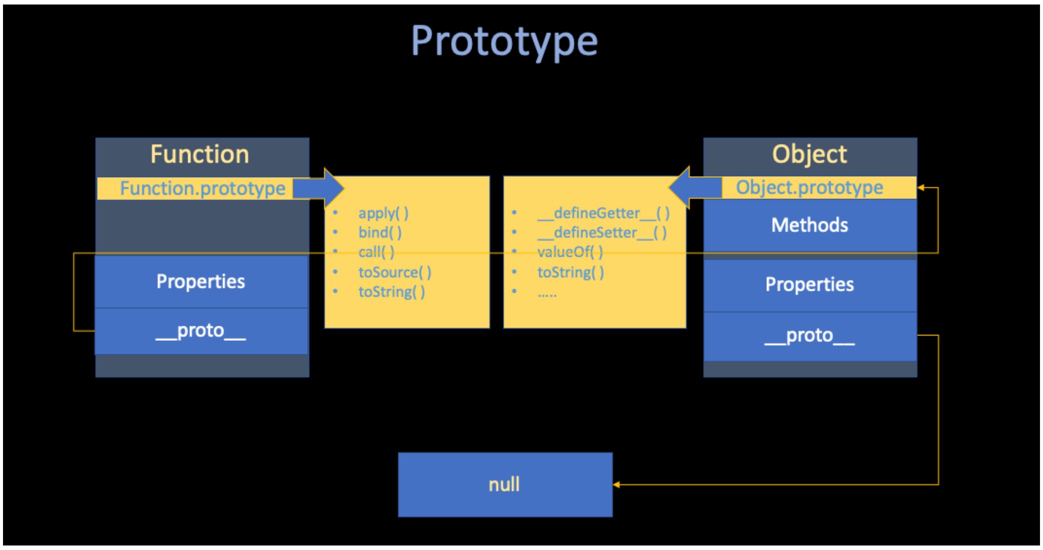
```
> point3D.toString
<· ƒ toString() { [native code] }
```

2. Function

Here comes a question:
We haven't defined any method of object `point3D`. How does this object have `toString()` method?
From the prototype chain we mentioned above. We can know `toString()` method must be defined in the prototype chain of object `point3D`.
So, where are the inherited properties and methods defined? If you look at the Object reference page, you'll see listed in the left hand inside a large number of properties and methods – many more than the number of inherited members we saw available on the `point3D` object. Some are inherited, and some aren't – why is this?

As mentioned above, the inherited ones are the ones defined on the **prototype** property – That is, the one that begin with `Object.prototype`, and not the ones that begin with just `Object`.

Here comes the formal definition of **Prototype:**
The Prototype property's value is just an Object, which is basically a bucket for storing properties and methods that want to be inherited by objects further down the prototype chain.



If you try the following in your console:

```
Object.prototype
```

You'll see many methods defined on Object's prototype property, which are then available on objects that inherit from object, as shown earlier.

You'll see other examples of **Prototype Chain Inheritance** all over JavaScript – try looking for the methods and properties defined on the prototype of the String, Date, Number, and Array global objects.

3. Class

Finally, let's step into **Class,** and try to understand how JavaScript implement **Class as an Object.**

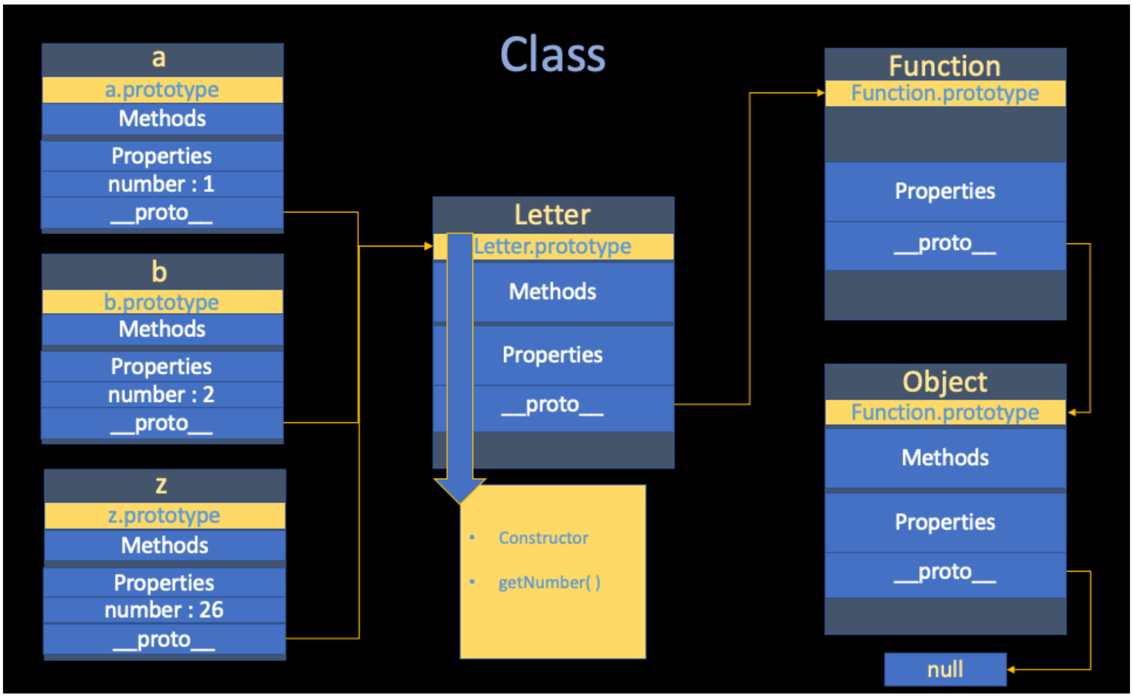
When several objects share the same initial state and behavior, they form a classification.

For user-continence (sometimes you don't want things to become cumbersome). Here comes the `class` keyword, which just a **syntactic sugar** (a construct which semantically does the same, but in much nicer syntactic form), and set the `prototype` implicitly.

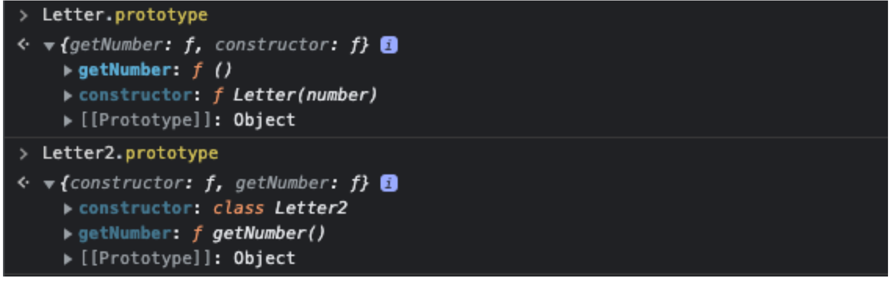
```
// Class
class Letter2 {
    constructor(number) {
        this.number = number;
    }
    getNumber() {
        return this.number;
    }
}

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
);
```

What the JavaScript interpreter does underneath – is just transform the **`class`** keyword into **`Constructor function + Prototype`** pair.
The actual work of the class `Letter2` must be exactly like this:

```
// Constructor Function
function Letter(number) {
    this.number = number;
};

// Set Prototype
Letter.prototype.getNumber = function() {
    return this.number;
}
```



And if you check both and prototype, you'll find that there is no such big difference – and of course, both are **Object.**



```
> Letter.prototype
<· ▾{getNumber: ƒ, constructor: ƒ} i
    ▸getNumber: ƒ ()
    ▸constructor: ƒ Letter(number)
    ▸[[Prototype]]: Object
> Letter2.prototype
<· ▾{constructor: ƒ, getNumber: ƒ} i
    ▸constructor: class Letter2
    ▸getNumber: ƒ getNumber()
    ▸[[Prototype]]: Object
```

4. Primitives

As I mentioned at the begin of this part of project:
All things in JavaScript can be presented as an Object, but there is something called Primitives that usually do not implement as an Object.

The **Primitives** is many things that you interact with regularly (e.g., String, Number), which most time when you manipulate them, you just care about the **Value** to do calculation, rather than those methods they have.

But **Primitives** do have **Object Wrappers**; these objects have and properties while the primitives do not. When you call a method of a primitive's type variable – It appear to have those methods.

For example, consider the following code:

```
var s = 'foo';
var sub = s.substring(1, 2); // sub is now the string "o"
```

That is because JavaScript interpreter silently create a **wrapper object** when code attempts to access any property of a **primitive**.
Behind the scenes, `s.substring(1, 2)` behaves as if it is performing the following (approximate) steps:

1. Create a wrapper `String` object, which is equivalent to using `new String(s);`
2. Call the method `substring()` with the appropriate parameters on the `String` object returned by step1;
3. Dispose of the `String` object;
4. Return the string (primitive) from step 2.

The following code prove this **"Wrapper Object Process"** exists:

```
1.  var i = 12;
2.  i.p1 = 13;
3.
4.  console.log(i);    // 12
5.  console.log(i.p1); // undefined
```

If we treat variable `i` as an `Integer` object instead of a primitive, we cannot explain why we cannot access `i.p1` in line 5 after we set this property in line 2.

But with considering the **Wrapper**: while we assign the property `i` to primitives in line 2.
It just create an `Integer` wrapper, and then **dispose** this object. And then in line 5, when we want to access `i.p1`, it just create another new `Integer` wrapper with no property called `p1`

In other words, we cannot retrieve them.