Notes on prototype and __proto__

__proto__

Every JavaScript object has a property called `__proto__`. Example:

```
var obj = {a: 10};
console.log(obj);            // { a: 10 }
console.log(obj.__proto__); // {}
```

The example above shows that `__proto__` of `obj` is defined but empty.

When we try to access the member of an object (like `object.member`), JavaScript looks up for the member in `__proto__` of the object if the member is not found in the object itself (like `object.__proto__.member`). If the member is still not found in `object.__proto__`, then it tries for `object.__proto__.__proto__.member` till either: it is found or the latest `.__proto__` itself is `null`. This explains why JavaScript is said to support prototypal inheritance out of the box.

```
var obj = {
    a: 10
};
obj.__proto__.a = 20;

console.log(obj.a); // 10

delete obj.a;

console.log(obj.a); // 20
```

prototype

Every `function f` in JavaScript also has a property called `prototype`, which has a property `constructor` which points to `f` itself. Example:

```
function MyFun() {}
```

```
console.log(MyFun);                                // [Function: MyFun]
console.log(MyFun.__proto__);                      // [Function]
console.log(MyFun.prototype);                      // MyFun {}
console.log(MyFun.prototype.constructor);          // [Function: MyFun]
console.log(MyFun.prototype.constructor === MyFun); // true
```

__proto__ and inheritance

Because JavaScript tries to fetch member from __proto__ recursively, __proto__ is useful to implement prototypical inheritance. A simple way to implement inheritance would be to assign <Base>.prototype to __proto__.__proto__ to an object which wants capabilities of <Base>. Example:

```
// We have two classes (function in pure JavaScript) Animal, and Bird
function Animal() {}
Animal.prototype.walk = function () { console.log("Walk"); }

function Bird() {}
Bird.prototype.fly = function () { console.log("Fly"); }

var bird = new Bird();
bird.fly();          // Fly
bird.walk();         // Throws error.

// However, if we do the following, then...
bird.__proto__.__proto__ = Animal.prototype;
// ... it works.
bird.walk(); // Walk

console.log("Is bird a Bird? " + (bird instanceof Bird) + ", Is bird an
Animal? " + (bird instanceof Animal));  // Is bird a Bird? true, Is bird an
Animal? true
```

We note here that fly is defined on Bird.prototype, and we access that from bird object. This is possible as fly is made available to bird.__proto__, by new Bird().

In fact, var bird = new Bird() assigns Bird.prototype to bird.__proto__. In JavaScript, this makes all members defined on class level (i.e. Type.prototype, here Bird.prototype) to the instances of that class (Refer this).

The fact that bird.__proto__ === Bird.prototype enables us to write the inheritance for the Bird class from Animal class more concisely, as follows.

```
// We have two classes (function in pure JavaScript) Animal, and Bird
function Animal() {}
Animal.prototype.walk = function () { console.log("Walk"); }

function Bird() {}
Bird.prototype.fly = function () { console.log("Fly"); }
Bird.prototype.__proto__ = Animal.prototype;  //<-- Nice!

var bird = new Bird();
bird.fly();         // Fly
bird.walk();        // Walk

console.log("Is bird a Bird? " + (bird instanceof Bird) + ", Is bird an
Animal? " + (bird instanceof Animal));  // Is bird a Bird? true, Is bird an
Animal? true
```

TODO: write about inheriting the instance level.

On ground level that's all we need to implement inheritance in JavaScript.

Note: From MDN:

> The instanceof operator tests whether an object in its prototype chain has the prototype property of a constructor.

Object.create and inheritance

However, as the property __proto__ is not *conveniently* available (for example in IDE and such), another way to write the same thing would be to use Object.create.

The syntax of Object.create is as follows:

```
Object.create(proto[, propertiesObject])
```

Object.create creates an object, prototype of which is of type as provided by proto. By using the second and optional propertiesObject additional properties to be added to the newly created object can be specified. Though, it can be useful to provide closure, we are skipping that discussion for now (Refer this).

Example:

```
// create an object that does not inherit from anything.
var obj = Object.create(null);
console.log(obj.__proto__); // undefined

// create derv that inherit from base
var base = { b: "base" };
var derv = Object.create(base, {
    d: {
        get: function () { return "derived"; }
    }
});

console.log(derv.d);                    // derived
console.log(derv.b);                    // base
console.log(derv.__proto__ === base);   // true
```

So, how Object.create can be used to implement inheritance? Well, we know that for inheritance we need Derived.prototype.__proto__ = Base.prototype. From the above example, we can see that Object.create assigns base to derv.__proto__. Then to implement inheritance with Object.create, we simply need to do Derived.prototype = Object.create(Base.prototype).

However, there are couple of things to note here. We'll discuss these with our working example of Animal and Bird.

```
function Animal() {}
Animal.prototype.walk = function () { console.log("Walk"); }

function Bird() {}
Bird.prototype.fly = function () { console.log("Fly"); }

// inherit from Animal
Bird.prototype = Object.create(Animal.prototype);

// now lets create a Bird object.
var bird = new Bird();

// and check for type of bird
console.log("Is bird a Bird? " + (bird instanceof Bird) + ", Is bird an
Animal? " + (bird instanceof Animal));           // Is bird a Bird? true, Is
bird an Animal? true // works!
bird.walk();        // Walk
bird.fly();         // TypeError: bird.fly is not a function // Wait... what?
```

So we see that in the above example, bird.fly() does not work. The reason for that is very simple. fly is defined on Bird.prototype. However, after defining fly on Bird.prototype, Bird.prototype was replaced, and thus, the fly is not accessible any more. So the trick is to declare any member on Derived.prototype after Derived.prototype is assigned Object.create(Base.prototype). So, the correction for this problem would be as follows.

```
// Omitting Animal for brevity
function Bird() {}
Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.fly = function () { console.log("Fly"); }

var bird = new Bird();
console.log("Is bird a Bird? " + (bird instanceof Bird) + ", Is bird an
Animal? " + (bird instanceof Animal));           // Is bird a Bird? true, Is
bird an Animal? true
bird.walk();        // Walk
bird.fly();         // Fly // works!
```

However, it still has one caveat though. Remember that `prototype` of every `function` has a property called `constructor`, which points to the function itself? So, from MDN we know that `prototype.constructor`

> Returns a reference to the Object constructor function that created the instance object. Note that the value of this property is a reference to the function itself...

So, lets see what happens to `prototype.constructor` with this implementation of inheritance.

```
// Omitting more code for brevity
function Bird() {}
Bird.prototype = Object.create(Animal.prototype);

var bird = new Bird();
console.log(bird.constructor); // [Function: Animal] // Why?
```

In the example we see that `bird.constructor` refers to `Animal`. So, this is obvious as `Bird.prototype = Object.create(Animal.prototype);` also assigns `Animal.prototype.constructor` (which is `Animal`) to `Bird.prototype.__proto__.constructor`. So, if we leave our inheritance implementation at this, we violate the definition of `prototype.constructor`. Then, it is a good practice to restore the `constructor`.

```
// Omitting more code for brevity
function Bird() {}
Bird.prototype = Object.create(Animal.prototype);
Bird.prototype.constructor = Bird;

var bird = new Bird();
console.log(bird.constructor); // [Function: Bird]
```

`extends` in TypeScript and inheritance

TODO