The following four lines are enough to confuse most JavaScript developers:

```
Object instanceof Function//true
```

```
Object instanceof Object//true
```

```
Function instanceof Object//true
```

```
Function instanceof Function//true
```

Prototype in JavaScript is one of the most mind-boggling concepts, but you can't avoid it. No matter how much you ignore it, you will encounter the prototype puzzle during your JavaScript life.

So let's face it head-on.

Starting with basics, there are following data types in JavaScript:

1. undefined

2. null
3. number
4. string
5. boolean
6. object

First five are primitive data types. These store a value of their type such as a boolean, and can be true or false.

The last "object" is a reference type which we can describe as a collection of key-value pairs (but it is much more).

In JavaScript, new objects are made using **Object constructor function** (or object literal `{}`) which provides generic methods like `toString()` and `valueOf()`.

Functions in JavaScript are special objects which can be "**called**". We make them and by using the **Function constructor function** (or function literal). The fact that these **constructors** are objects as well as function has always confused me, much in the same way the chicken-egg riddle confuses everyone.

Before starting with Prototypes, I want to clarify that there are two prototypes in JavaScript:

1. **prototype**: This is a special object which is assigned as property of any function you make in JavaScript. Let me be clear here, it is already present for any function you make, but not mandatory for internal functions provided by JavaScript (and function returned by `bind`). This `prototype` is the same object that is pointed to by the `[[Prototype]]`

(see below) of the a newly created object from that function (using `new` keyword).

2. **[[Prototype]]:** This is a somehow-hidden property on every object which is accessed by the running context if some property which is being read on the object is not available. This property simply is a reference to the `prototype` of the function from which the object was made. It can be accessed in script using special **getter-setter** (topic for another day) called `__proto__`. There are other new ways to access this prototype, but for sake of brevity, I will be referring to `**[[Prototype]]**` using `__proto__`.

```
var obj = {}var obj1 = new Object()
```

The above two statements are equal statements when used to make a new object, but a lot happens when we execute any of these statements.

When I make a new object, it is empty. Actually it is not empty because it is an instance of the `Object` constructor, and it inherently gets a reference of `prototype` of `Object,` which is pointed to by the `__proto__` of the newly created object.

```
> var obj = {}
< undefined
> obj
< ▼ Object {}
    ▼ __proto__: Object
      ▶ constructor: function Object()
      ▶ hasOwnProperty: function hasOwnProperty()
      ▶ isPrototypeOf: function isPrototypeOf()
      ▶ propertyIsEnumerable: function propertyIsEnumerable()
      ▶ toLocaleString: function toLocaleString()
      ▶ toString: function toString()
      ▶ valueOf: function valueOf()
      ▶ __defineGetter__: function __defineGetter__()
      ▶ __defineSetter__: function __defineSetter__()
      ▶ __lookupGetter__: function __lookupGetter__()
      ▶ __lookupSetter__: function __lookupSetter__()
      ▶ get __proto__: function __proto__()
      ▶ set __proto__: function __proto__()
```

__proto__ of obj

If we look at the `prototype` of `Object` constructor function, it looks the same as the `__proto__` of `obj`. In fact, they are two pointers referring to the same object.

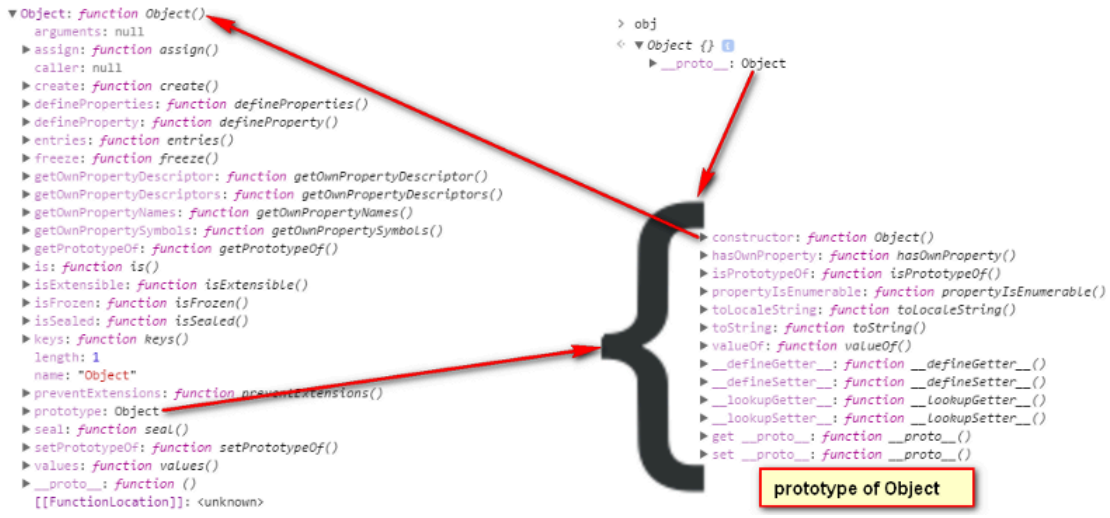
```
> Object.prototype
< ▼ Object {__defineGetter__: function, __defineSetter__: function, h
    ▶ constructor: function Object()
    ▶ hasOwnProperty: function hasOwnProperty()
    ▶ isPrototypeOf: function isPrototypeOf()
    ▶ propertyIsEnumerable: function propertyIsEnumerable()
    ▶ toLocaleString: function toLocaleString()
    ▶ toString: function toString()
    ▶ valueOf: function valueOf()
    ▶ __defineGetter__: function __defineGetter__()
    ▶ __defineSetter__: function __defineSetter__()
    ▶ __lookupGetter__: function __lookupGetter__()
    ▶ __lookupSetter__: function __lookupSetter__()
    ▶ get __proto__: function __proto__()
    ▶ set __proto__: function __proto__()
```

prototype of Object

```
obj.__proto__ === Object.prototype//true
```

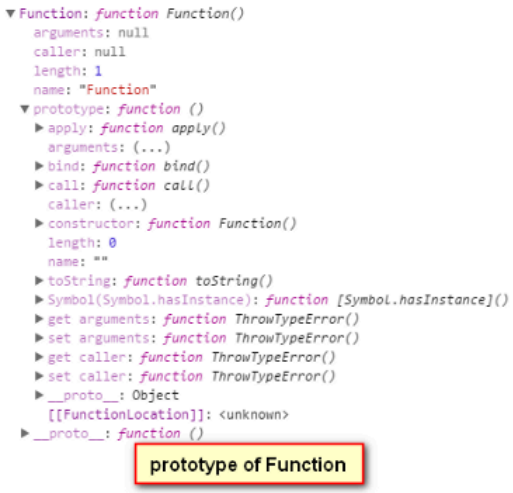
Every `prototype` of a function has an inherent property called `constructor` which is a pointer to the function itself. In the case of `Object` function, the `prototype` has `constructor` which points back to `Object`.

```
Object.prototype.constructor === Object//true
```



In the picture above, the left side is the expanded view of the `Object` constructor. You must be wondering what are all these other functions over it. Well, functions are **objects**, so they can have properties over them as other objects can.

If you look closely, the `Object` *(on left)* itself has a `__proto__` which means that `Object` must have been made from some other constructor which has a `prototype.` As `Object` is a function object, it must have been made using `Function` constructor.



`__proto__` of `Object` looks same as `prototype` of `Function` ._.When I check the equality of both, they turn out to be the same objects.

```
Object.__proto__ === Function.prototype//true
```

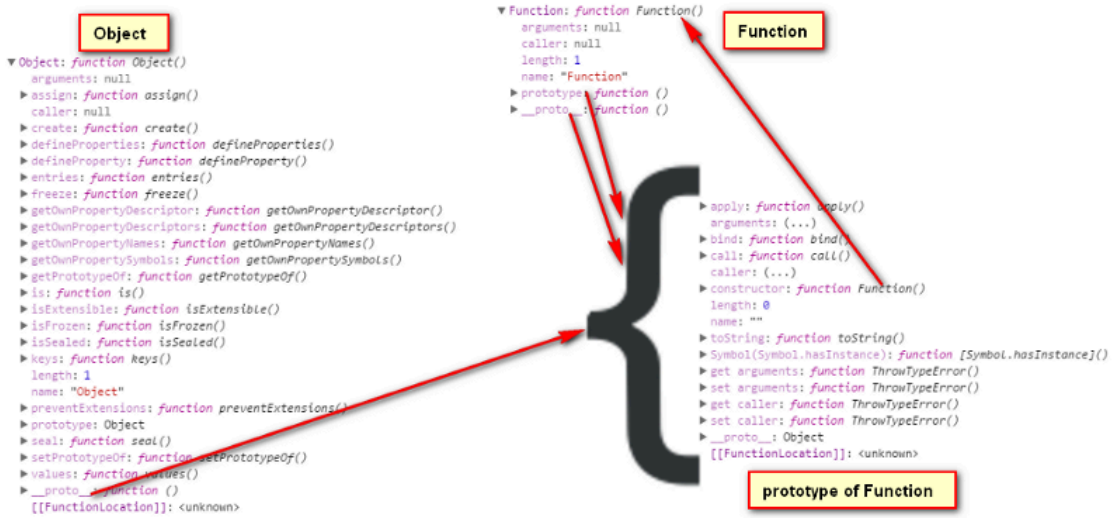
If you look closely, you will see the `Function` itself has a `__proto__` which means that `Function` constructor function must have been made from some constructor function which has a `prototype`. As `Function` itself is a **function**, it must have been made using `Function` constructor, that is, itself. I know that sounds weird but when you check it, it turns out to be true.

The `__proto__` of `Function` and `prototype` of `Function` are in fact two pointers referring to the same object.

```
Function.prototype === Function.__proto__\\true
```

As mentioned earlier, the `constructor` of any `prototype` should point to the function that owns that `prototype.` The `constructor` of `prototype` of `Function` points back to `Function` itself.

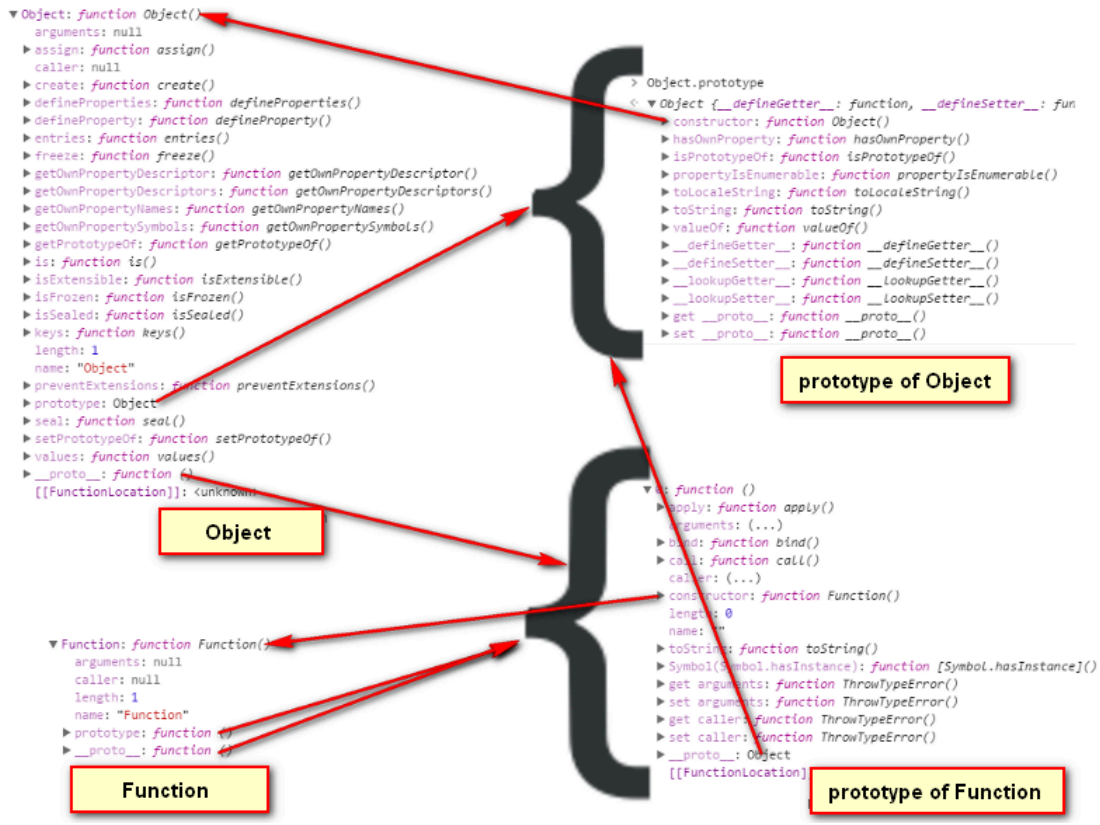
```
Function.prototype.constructor === Function\\true
```



Again, the `**prototype**` of `**Function**` has a `__proto__` .Well, that's no surprise… `prototype` is an object, it can have one. But notice also that it points to the `prototype` of `Object` .

```
Function.prototype.__proto__ == Object.prototype\\true
```

So we can have a master map here:




```
instanceof Operatora instanceof b
```

The `instanceof` operator looks for the object `b` pointed to by any of the `constructor`(s) *of* chained `__proto__` on `a`. Read that again! If it finds any such reference it returns `true` else `false`.

Now we come back to our first four `instanceof` statements. I have written corresponding statements that make `instanceof` return `true` for the following:

```
Object instanceof FunctionObject.__proto__.constructor === Funct
```

```
Object instanceof ObjectObject.__proto__.__proto__.constructor ==
```

```
Function instanceof FunctionFunction.__proto__.constructor === Fu
```

```
Function instanceof ObjectFunction.__proto__.__proto__.constructo
```

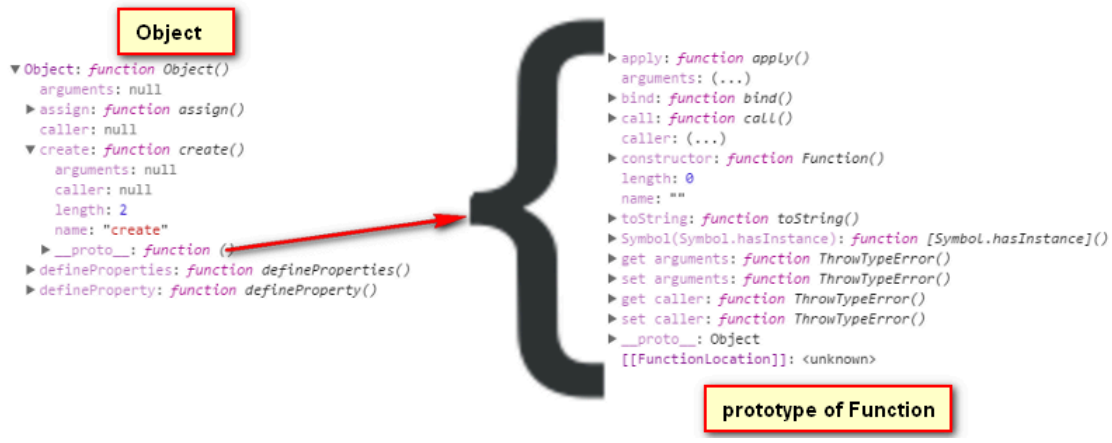
Phew!! Even spaghetti is less tangled, but I hope things are clearer now.

Here I have something that I did not pointed out earlier that `prototype` of `Object` doesn't have a `__proto__`.

Actually it has a `__proto__` but that is equal to `**null**`. The chain had to end somewhere and it ends here.

```
Object.prototype.__proto__\\null
```

Our `Object`, `Function`, `Object.prototype` and `Function.prototype` also have properties which are functions, such as `Object.assign`, `Object.prototype.hasOwnProperty` and `Function.prototype.call`. These are internal functions which do not have `prototype` and are also instances of `Function` and have a `__proto__` which is a pointer to `Function.prototype`.




```
Object.create.__proto__ === Function.prototype\\true
```

You can explore other constructor functions like `Array` and `Date`, or take their objects and look for the `prototype` and `__proto__`. I'm sure you will be able to make out how everything is connected.

Extra queries:

There's one more question that bugged me for a while: Why is it that `prototype` of `Object` is **object** and `prototype` of `Function` is **function object**?

Here is a good explanation for it if you were thinking the same.

Another question that might be a mystery for you until now is: How do primitive data types get functions like `toString()`, `substr()` and `toFixed()`? This is well explained **here**.

Using `prototype`, we can make inheritance work with our custom objects in JavaScript. But that is a topic for another day.

Thanks for reading!