javascript prototypes and classes

Steven Brawer
May 11, 2017

introduction 2
prototypes of simple, non-function objects 3
the prototype chain 5
manipulating prototypes 6
prototypes for functions 9
the prototype chain 9
function.prototype 12
function constructors 14
classes - 1 - the simplest way 16
prototypes of instances and classes 17
subclass 20
classes - 2 - the prototype way 22
subclassing 26
instanceof 28
classes 3 - lower-level implementation 29

introduction

Javascript is called a **prototypal language**, because its key data structure is the **prototype**. Of course there is a lot more to javascript, but prototypes are its distinguishing feature. Prototypes enable behaviors of object oriented programming (OOP) which are similar to but can be different from the usual class-based approach.

The crucial data structure in OOP is the class, from which instances are created. In vanilla OOP, classes implement the concept of data and function encapsulation. Javascript can be used to implement the basic concepts of class and (single) inheritance, but it is more general than that.

- **Javascript ES6 has syntactic sugar for classes and subclasses.** These were added to aid developers who are experienced with class-based programming with languages such as C++, Java, Smalltalk, Python but not familiar with prototypes.
- **Javascript provides lower-level tools to create different styles of OOP.** In fact, until ES6, these tools were all that were available.
- **Javascript can be used for functional programming,** which in a sense is a severe type of encapsulation.

This document explores how prototypes are used in implementing OOP. It starts off with a description of the vanilla behavior of prototypes. Then it discusses ES6 class operators, showing how prototypes implement the usual class paradigm. It then presents two programs in which the equivalent of classes are created at a lower level.

This document refers to ES6 and is intended for developers who have used javascript a fair amount but still have an uncertain understanding of prototypes.

In many cases, I show a screen copy of *console.log(...)* output. These are all from **Chrome**.

I use the word *field* as a synonym for *property*. ie,

"field" = "property".

prototypes of simple, non-function objects

Start with the simple object

```
let obj = {x: 1,y: "now is the time"};
```

This object has two **properties** - *x* and *y*. These are **own-properties**. None of its properties are functions, so we can say that it has no **own-functions.**

By default, every object created by literals, such as *obj* above, is created by the runtime with a **hidden pointer** to an object which can be called a **universal prototype** That prototype is itself an object with own-properties, and it endows any object (such as *obj* above) with a basic functionality, implemented by own-functions of the prototype. We could say that *obj* **has** a prototype, but saying that *obj* **points to** the prototype is more accurate. **The universal prototype is a singleton prototype object which is shared among all objects.**

Every object created as above has a hidden pointer to the exact same prototype. The prototype is an object. You can access this prototype in two ways: the new way and a deprecated way.

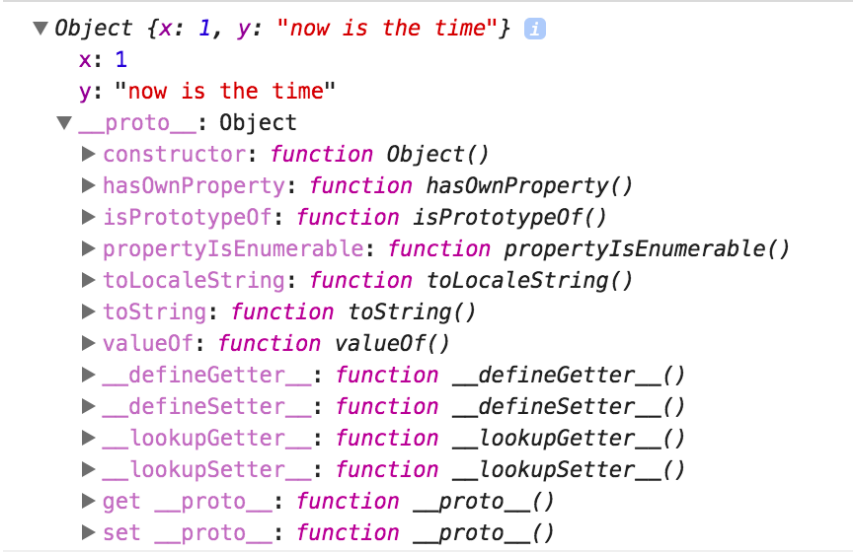
```
let p2 = Object.getPrototypeOf(obj);      // new way
let p3 = Object.__proto__;                // deprecated in ES6
p2 === p3;                                // true
```

The variable *p2* is (a pointer to) the prototype of *obj*, and *p3* is (a pointer to) the same identical object as *p2*. So *obj.__proto__* does the same thing as *Object.getPrototypeOf(obj),* but *__proto__* is deprecated. Nonetheless, it is still supported in ES6. Apparently it will be supported forever, for legacy purposes.

The prototype can be output to the console. The expression

```
console.log(p2);
```

gives in Chrome



The universal prototype is the object under *__proto__*.

As an example of the functionality of *obj* endowed by the prototype is the function *hasOwnProperty(prop)*, which allows the determination of whether a property *prop* is part of *obj*. Thus

```
obj.hasOwnProperty("x");   //true
obj.hasOwnProperty("z");   // false;
```

In this case, *x* is an *own-property* of *obj* while *z* is not an *own-property* of obj.

We call *__proto__* a **hidden pointer** because of the contradictory behavior below:

```
obj.hasOwnProperty("__proto__");          // false
obj.__proto__;                            // the prototype object
```

The function *hasOwnProperty* is not in *obj* (which has only two properties, x and y), it is in the universal prototype. So

```
console.log(p2.hasOwnProperty("hasOwnProperty"));  // true
```

the prototype chain

Consider

```
let obj1 = {a: 1, b: 2};
let obj2 = {a: 5,b:6};

let pr1 = Object.getPrototypeOf(obj1);
let pr2 = Object getPrototypeOf(obj2);
pr1 === pr2;                                    // true
```

The object *pr1* is the prototype of *obj1* and *pr2* is the prototype of *obj2*, and these prototypes are identical - they are both the *universal prototype* object. (Of course, *obj1* and *obj2* are not identical, so *obj1 === obj2* is false.)

Now we **create a property z in the prototype.**

```
console.log( obj1.z,  obj2.z );        // undefined undefined
pr1.z = "hello";                       // add z to universal prototype
console.log(obj1.z, obj2.z);           // hello hello
```

Now both *obj1* and *obj2* appear to have a property *z* with exactly the same value. However, they do not.

```
obj1.hasOwnProperty("z"); // false
obj2.hasOwnProperty("z"); // false
```

The property is in the prototype.

```
pr1.hasOwnProperty("z");  // true
```

Since *obj1* and *obj2* point to the same identical prototype, they both appear to have property *z,* with exactly the same value in each. Here's how this works. First, we note the following:

```
let protOfProt = Object.getPrototypeOf(pr1);
console.log(protOfProt);                        // null
```

So in this case, the universal prototype has *null* as its prototype. The universal prototype does not itself have a prototype in the default situation (it can be given a prototype, though generally there is not much point to that).

When the statement *obj1.z* is executed:

- first the own-properties of obj1 (in this case, *x* and *y*) are examined.
- If the property z is there, the value of *z* is returned.
- If it is not there, then the prototype of the object is examined.
- If that prototype has the property *z*, then the value of *z* is returned.
- If it is not there, then the prototype of the prototype is examined for the property *z*.
- If it is found, the value is returned.
- If not there, keep going until a prototype is *null*.
- In that case *z* is undefined.

This is called the **prototype chain**.

Since the identical prototype is pointed to by both *obj1* and *obj2*, they both return the same value of z. Thus

```
pr1.z = "goodbye";
console.log( obj1.z,  obj2.z );                    // goodbye goodbye
```

Now consider:

```
obj1.z = "my own z";
console.log( obj1.z,  obj2.z );          // "my own z"   hello
```

The statement *obj1.z = "..."* gives *obj1* the own-property *z*. In this case, when *obj1.z* is executed, it is found that *obj1* has its own-property *z*, so that value is returned. However, *obj2* does not have own-property *z*, so the value of *z* from its prototype is returned.

manipulating prototypes

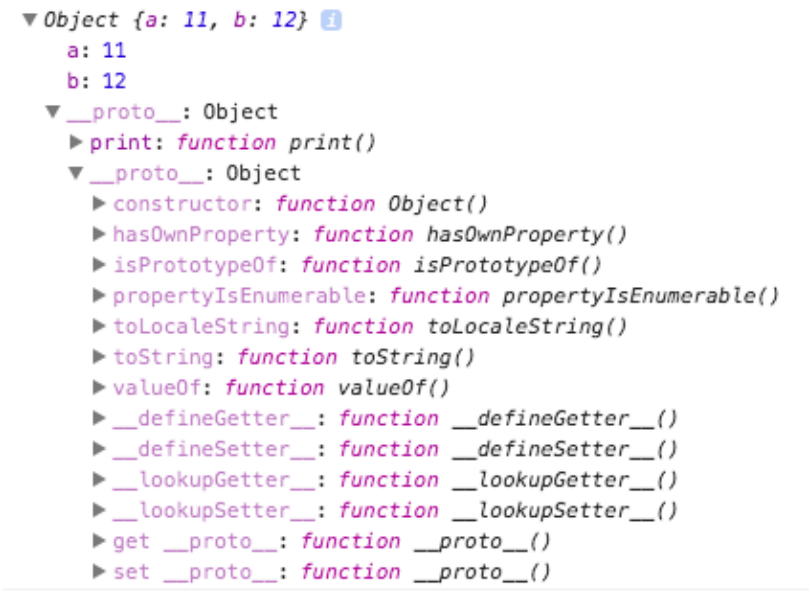
The prototype of an object can be modified (by adding or deleting properties). Consider

```
function print(){console.log(this.a,  this.b);};
let obj3 = {a:11,b:12};
let prot = {print: print};

Object.setPrototypeOf(obj3, prot);          // makes prot the prototype of obj3
obj3.print();                                                  // 11  12
console.log(obj3.hasOwnProperty("a"));                         // true
console.log(obj3.hasOwnProperty("print");                      // false
console.log(obj3.__proto__.hasOwnProperty("print");   // true
```

The expression *Object.setPrototypeOf(obj3, prot);* makes *prot* the prototype of *obj3*. The question is: where did the method *obj3.hasOwnProperty()* come from? In the above code, we have replaced the existing prototype (the universal prototype) of *obj3* with an object *prot* with only the property *print*.

The answer is this. When *prot* is created, it is automatically given the universal prototype as the property of its hidden pointer *__proto__* in *prot*.

```
console.log(obj3);
```



As noted above, the universal prototype is generated automatically. So now *obj3* has *prot* as its prototype while *prot* has the universal prototype as its (*prot's*) prototype. The universal prototype in turn has *null* as its prototype. Therefore, when the runtime is looking for *obj3.hasOwnProperty(),* first it looks in *obj3*, then in the prototype *prot* of *obj3*. The method is in the prototype of the prototype of *obj3*, so it is then executed.

Note the form of the print() function of *prot*. The *"this"* refers to *obj3*. In fact, if we modify print to

```
function print(){console.log(this, this.a, this.b);};
```

The result is

```
Object {a: 11, b: 12} 11 12
```

so that *this.a* in the print function really means *obj3.a*.

In general, for a function in a prototype, "this" refers to the base object, but the search for properties goes through the prototypes as usual. Let's modify the above example to make that clear.

```
function print(){console.log(this, this.a,this.b, this.z);};
let obj3 = {a:11,b:12};
let prot = {print: print};

Object.setPrototypeOf(obj3, prot);
let univ = Object.getPrototypeOf(prot);    // universal prototype
univ.z = "I am z";                         // z is property of universal prototype
obj3.print();                              11 12 "I am z"
```

So we have:

obj3 - (base object) own-properties *a,b*. Points to
prot - prototype of *obj3*, property *print(this....)*, where "this" refers to obj3. Points to
univ - modified universal prototype of *prot*, with property *z. P*oints to
null

In the above example we have modified the universal prototype, so all objects either alredy created or to be created will have the property *z*. Generally, this is considered a very bad idea.

prototypes for functions

The way in which prototypes work in functions is made obscure by the fact that a function has an own-property **prototype**, which non-function objects do not have. We have the following features of prototypes in functions:

- The prototype chain of functions is determined exactly as in non-function objects - by the expression *Object.getPropertyOf(anObject) or anObject.__proto__.*
- Every function has an own-property **prototype** which is used by the unary operator *new* and by the binary operator *instanceof*. **It is not in the prototype chain.**
- Non-function objects do not by default have the own-property *prototype*. Their prototype chain was discussed previously, and is the same as for function objects.
- The way in which functions and prototypes of functions are output by *console.log(....)* can be: uninformative, or misleading, or useful (at least for Chrome and Safari).

I now expand on these comments.

the prototype chain

Consider

```
function Foo(a,b){
        this.x = a;
        this.y = b;
        };
console.log(Foo);      // function Foo(a,b){this.x = a; this.y = b;}
```

This output is certainly not useful nor informative. To examine the own-properties of *Foo*, we may do the following:

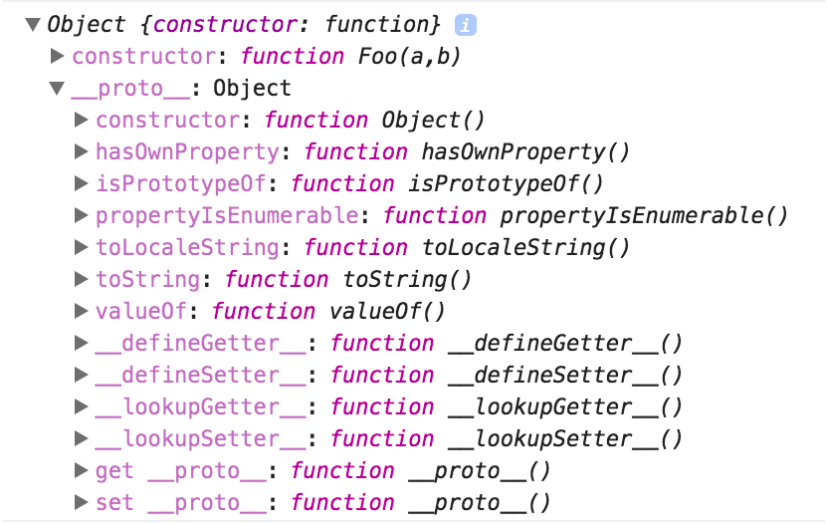
```
let keys = Object.getOwnPropertyNames(Foo);
console.log(keys); // ["length", "name", "arguments", "caller", "prototype"]
```

Note that the variables *x,y* are not included - these are not properties, as they would be for a non-function object.

We see that *Foo* has a property *prototype*.

```
let prot = Foo.prototype;
console.log(prot);
```

The output is (in Chrome)



We see that, according to the console output, *Foo.prototype* has two properties: *constructor* and the hidden pointer *__proto__*. I will discuss the constructor below. But, just as for non-function objects, *__proto__* is not a normal own-property.

```
console.log(Foo.prototype.hasOwnProperty("__proto__"); // false
```

On the other hand

```
console.log(Foo.prototype.__proto__);
```

gives the output shown in the figure above. *Foo.prototype.__proto__* is (a pointer to) the *universal prototype*, the same as discussed above in the context of non-function objects. We can see this as follows. First we create a property **tag** in *Foo.prototype.__proto__*.

```
Foo.prototype.__proto__.tag = function() { console.log("tag"); };
console.log(Foo.prototype.__proto__.tag);          // function () { console.log("tag"); }
console.log(Foo.prototype.__proto__.hasOwnProperty("tag"));     // true
```

So *tag* is really there. Now consider the universal prototype of a non-function object

```
let obj = {};
let prot = Object.getPrototypeOf(obj);                  // universal prototype
console.log(prot.hasOwnProperty("tag"));          // true
```

So we have created a property *tag* in the universal prototype by assigning it in Foo.prototype.__proto__.

Now we ***start from scratch with Foo***, so none of the changes above are incorporated. We show that *Foo.prototype.__proto__* is not in the prototype chain of Foo. (The purpose of *Foo.prototype* is discussed later in this chapter.)

We note that the function *Foo.call* exists, even though neither *Foo.prototype* nor *Foo.prototype.__proto__* includes this function.

```
console.log(Foo.prototype.call);                // undefined
console.log(Foo.prototype.__proto__.call); // undefined
console.log(Foo.call);                          // function call() { [native code] }
console.log(Foo.hasOwnProperty("call")); // false
```

So *call* really exists, but is not an own-property of *Foo*. For example, we can use this function in one of the most important patterns in javascript.

```
let obj = {};
Foo.call(obj,4,5);
console.log(obj);       // Object {x: 4, y: 5}
```

So the function *call* is somehow in the prototype chain, but not in a chain involving *Foo.prototype*. But where is it? One might try the obvious first thing.

```
let prot = Object.getPrototypeOf(Foo);
console.log(prot);                              //function () { [native code] }
```

This is not useful. We point out that *Foo* must have the hidden pointer *__proto__,* so *Foo.__proto__* is an object. Therefore

```
let prot = Object.getPrototypeOf(Foo);              // gets Foo.__proto__
let keys = Object.getOwnPropertyNames(prot);
console.log(keys);
        // ["length", "name", "arguments", "caller", "apply", "bind", "call", "toString",
"constructor"]
console.log(prot.hasOwnProperty("call"));       // true
```

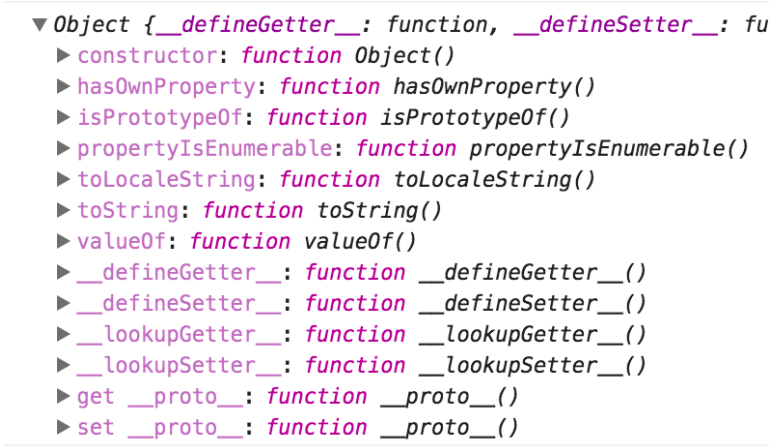
Here is where *call()* resides. We see that *Foo.prototype* and the prototype *prot* above are not the same object.

```
console.log(prot === Foo.prototype);        // false
```

What about other useful functions, such as *hasOwnProperty*.

```
let protOfProt = Object.getPrototypeOf(prot);
console.log(protOfProt);
```

The output is



This is the universal prototype. Note that

```
console.log(protOfProt === Foo.prototype.__proto__);      // true
console.log(protOfProt.hasOwnProperty("hasOwnProperty")); // true
```

So **the universal prototype appears in several places** in *Foo,* which could be confusing.

We have shown the following for functions:

- **Foo.prototype is not in the prototype chain.**
- **Object.getPropertyOf(...) gives the prototype chain, just as for non-function objects.**

Note that *Foo.__proto__* does not show up in the output *console.log(Foo)*. This is one example of where the browser output can be misleading as well as not useful.

function.prototype

The **prototype** property of a function is used by the **new** operator when creating new objects. The object pointed to by the property *prototype* becomes the prototype of the new object, and is retrieved by *Object.getPrototypeOf(newObject)*. Here's an example which illustrates the point.

```
function Foo(a,b){
        this.x = a;
        this.y = b;
        };
Foo.prototype.tag = function(){ console.log("tag"); };
let newObj = new Foo(10,15);
console.log(newObj);
```

Here is the output.

The variables *x,y* become the own-properties of *newObj*. The object *Foo.prototype*, which includes function *tag()* and the hidden pointer *__proto__* (which points to the universal prototype), becomes the prototype of *newObj*.

```
let prot = Object.getPrototypeOf(newObj);
console.log(prot === Foo.prototype);          // true
newObj.tag();                                 // tag
```

However

```
Foo.tag();                    // Error, Foo.tag is not a function
```

The reason for this, we should now understand, is because *Foo.prototype* is not in the prototype chain of *Foo*.

Note that ***prot* obtained from *newObj* is identical to *Foo.prototype,* not a copy.** They are the same object. The operator *new* makes *newObj.__proto__* point to *Foo.prototype*. Since *Foo.prototype* already has a pointer *__proto__* to the universal prototype, the universal prototype comes along for the ride.

The above example also shows how we can insert the function *tag* into *Foo.prototype*, and this function is inherited by objects created with the *new* operator, even after they are created. We can also do the flip side - insert a function into the prototype of *newObj*, and it will be "inherited" by *Foo.prototype* itself, and then inherited by additional new objects as well as all existing objects created from Foo. This is because of the fact that

- **prototypes are singletons,**
- **are pointed to,**
- **and are not copies**.

```
let prot = Object.getPrototypeOf(newObj);
prot.bar = function(){console.log("bar"); // puts bar into Foo.prototype
                                          // newObj now has bar
console.log(Foo.prototype.bar);    // function (){console.log("bar")} "inherited" by Foo
newObj.bar();                      // bar
anotherObj = new Foo(20,25);
anotherObj.bar();                  // bar
```

In addition, *Foo.prototype* is used in the operator *instanceof*, as will be discussed when I discuss classes.

function constructors

Foo.prototype.constructor is used by the *new* operator to create new objects. I'm not going to say more about constructors, except to show how they really complicate the console output of functions.

A constructor is the same identical object as the function itself.

```
console.log(Foo === Foo.prototype.constructor)  // true
```

Because of this circularity, we can write things like

```
let x = Foo.prototype.constructor.prototype.constructor;
console.log(Foo === x);
```

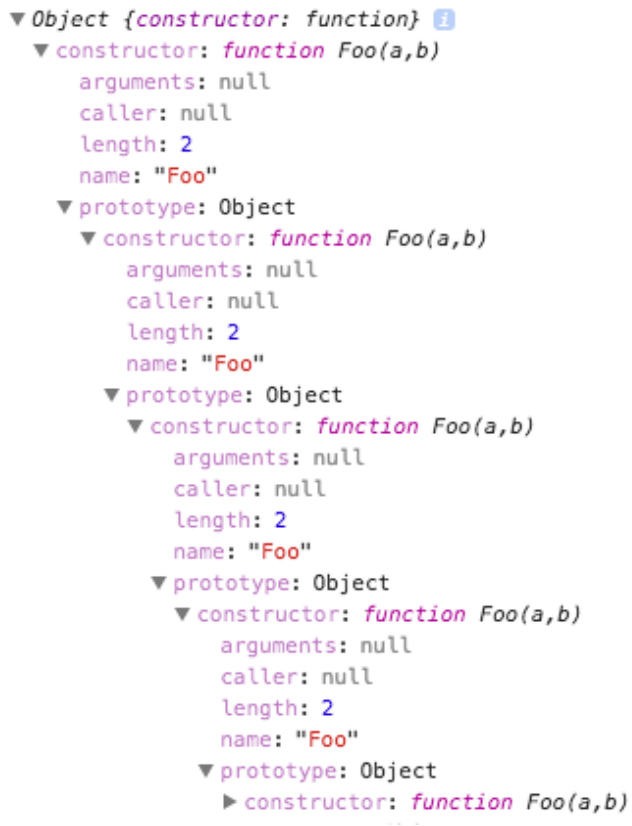
and in fact we have the ridiculous snippet:

```
x = Foo.prototype.constructor;
for(let i=0;i<10;i++) x = x.prototype.constructor;
console.log(Foo === x);        // true
```

It is presumably for this reason that the following output is so complicated.

```
console.log(Foo.prototype);
```

If we expand the output, seems to go on forever, as below.


▼ Object {constructor: function}
▼ constructor: function Foo(a,b)
arguments: null
caller: null
length: 2
name: "Foo"
▼ prototype: Object
▼ constructor: function Foo(a,b)
arguments: null
caller: null
length: 2
name: "Foo"
▼ prototype: Object
▼ constructor: function Foo(a,b)
arguments: null
caller: null
length: 2
name: "Foo"
▼ prototype: Object
▼ constructor: function Foo(a,b)
arguments: null
caller: null
length: 2
name: "Foo"
▼ prototype: Object
▶ constructor: function Foo(a,b)

and on and on.

classes - 1 - the simplest way

We now embark on an exploration of how Javascript implements OOP concepts. We start with the ES6 class operators, the simplest and most direct way. Even here, javascript makes use of prototypes under the covers, though prototypes are not explicit in class creation. Many people consider the operators *class* and *extends* syntactic sugar, though they are convenient and don't seem to forestall other ways of doing things.

```
//==================  "class" Smaller ================


class Smaller{
        constructor(a,b){
                console.log("constructor Smaller");
                this.thing = a;         // own-property
                this.tag = b;           // own-property
        };

        print(){
                console.log(this.thing, this.tag);
        };

        addToThing(num){
                this.thing += num;
                return this.thing;
        };
};

//==================  "class" Bigger (subclass of Smaller) ================

class Bigger extends Smaller{
        constructor(a,b,c){
                super(a,b);                     // calls Smaller constructor
                console.log("constructor Bigger");
                this.stuff = c;                 // own-property
        };

        print(){
                console.log(this.thing, this.tag, this.stuff);
        };
};
```

We have base class *Smaller*. (Actually, a base class is always a subclass of *Object*.) The class *Bigger* is a subclass of *Smaller*. The constructor function is executed when we use *new* to

create an instance of the class. With the keyword *new*, the behavior of the function *Smaller* and the constructor are coordinated, so they work together to create an object.

In any event,

```
let small1 = new Smaller(1,"now is the time");
// outputs "constructor Smaller"
```

constructor() is just an ordinary function, and *Smaller* appears to be an ordinary function.However, with the keyword *class*, function *Smaller* should not be called directly. When we drill down in later sections, we will see how function *Smaller* can be used under the covers.

In the OOP terminology, *Smaller* is a **class** and *small1* is an **instance** of class *Smaller*. To javascript, they are both objects. The instance *small1* behaves in the manner expected from other OO languages.

```
console.log(small1 instanceof Smaller);          // true
small1.print();                                  // 1 "now is the time"
small1.addToThing(20);
small1.print();                                  // 21 "now is the time"
console.log(small1.hasOwnProperty("thing"));     // true
console.log(small1.hasOwnProperty("stuff"));     // false
```

Both *Smaller* and *small1* are ordinary javascript objects. The *instanceof* statement is really a shorthand for comparing the identities of prototypes of instances and classes, such as the prototypes of *small1* and *Smaller*, as we will see in the next chapter.

Before preceding, we create another instance of Smaller.

```
let small2 = new Smaller(500, "no more time");
// outputs "constructor Smaller"
small2.print();                          // 500 "no more time"
small2.addToThing(20);
small2.print();                          // 520 "no more time"
```

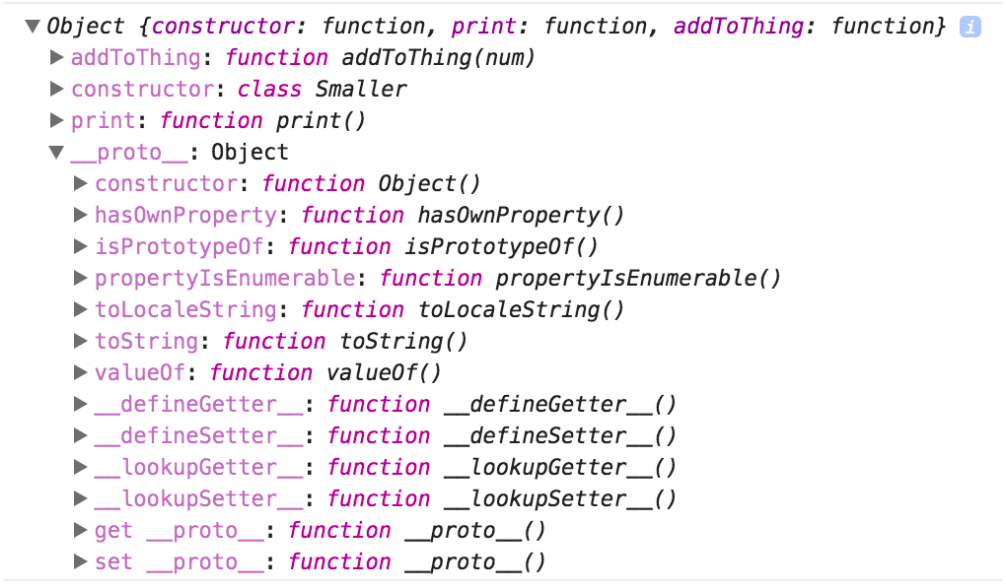
So we see that the values of the properties *thing* and *tag* are different in *small1* and *small2*, so these are own-properties. However, the methods *print()* and *addToThing()* are the same for both *small1* and *small2*, so we would expect these to be properties of a prototype. This is shown in the next section.

prototypes of instances and classes

Where do the functions *print* and *addToThing* fit in? Consider

```
console.log(Smaller.prototype);
```

The output is



We see the expected property *constructor* and hidden pointer *__proto__*, but also the *print* and *addToThing* functions. When we do

```
let small1 = new Smaller(....);
```

we expect *Smaller.prototype* to be the object pointed to by *small1.__proto__*, and this is indeed the case.

```
let prot = Object.getPrototypeOf(small1);          // created by new
prot.hasOwnProperty("print");                      // true
let univ = Object.getPrototypeOf(prot);            // universal prototype
univ.hasOwnProperty("hasOwnProperty");             // true

console.log(prot === Smaller.prototype);           // true
```

The expression

```
console.log(small1)
```

outputs the following in Chrome:

The base object has own-properties *tag* and *thing*, as we expect. The prototype of the base object has three properties: *print,(), addToThing()* and *constructor()* and hidden pointer *__proto__*. The prototype of that prototype is the universal prototype.

Even though *small1* is an instance of class *Smaller*, all prototype manipulations can be done with it. Not all OOP languages allow this sort of thing. But in javascript, the following is unremarkable.

```
let prot = Object.getPrototypeOf(small1);
prot.smallJunk = "I am smallJunk";
prot.printMore = function(){
                this.print();
                console.log(this.smallJunk);
                };
small1.printMore();                 // 21 "now is the time" \n "I am smallJunk"
small2.printMore();                 // 520 "no more time" \n "I am smallJunk
small1.smallJunk = "OWN smallJunk";
small1.printMore();                 // 21 "now is the time" \n "OWN smallJunk"
small2.printMore();                 // 520 "no more time" \n "I am smallJunk
```

subclass

Consider the following.

```
let big1 = new Bigger(1000,"much less time", [1,2,3,4]);
// outputs "constructor Bigger \n constructor Smaller"
big1.print();                                   // 1000 "much less time" [1,2,3,4]

console.log(big1 instanceof Bigger);            // true
console.log(big1 instanceof Smaller);           // true
console.log(small1 instanceof Smaller);         // true
console.log(small1 instanceof Bigger);          // false
```

We examine *big1*. The expression

```
console.log(big1)
```

gives, in Chrome:

The **inheritance** of *Bigger* from *Smaller* occurs because, for any instance of Bigger, *Smaller.prototype* is the prototype of *Bigger.prototype*, and this relation is preserved when the

big1 object is created with the operator *new*. Therefore, when invoking a method on *big1*, we have the steps:

- First look in the own-properties of *big1*. If it is there, execute it.
- If not, look in the prototype of *big1 (*which is *Bigger.prototype)*. If there, execute it.
- Next look in the prototype of the prototype of *big1* (which is Smaller.prototype). If it is there, execute it.
- Next look in the prototype of the prototype of the prototype of *big1* (which is the universal prototype). If it is there, execute it.
- The next prototype along the chain is null, so the function is undefined.

To add to class *Bigger* a function which invokes the *print* function of *Smaller (*which is *printSmaller)* add a function to class *Bigger*:

```
class Bigger extends Smaller{
        ....
        ....
printSuper(){ super.print(); };
}
```

then

```
big1.printSuper();              // 1000 "much less time"
```

One could do it this way with prototypes.

```
big1.__proto__.printSuper = printSuper;
function printSuper(){
        this.__proto__.__proto__.print.call(this);
}
big1.printSuper();                      // 1000 "much less time"
let big2 = new Bigger(50,"hurry", []);
big2.printSuper();                      // 50 "hurry"
```

The problem is, it is a javascript error to have the following function definition outside the class definition.

```
function printSuper(){
        super.print();
}
```

classes - 2 - the prototype way

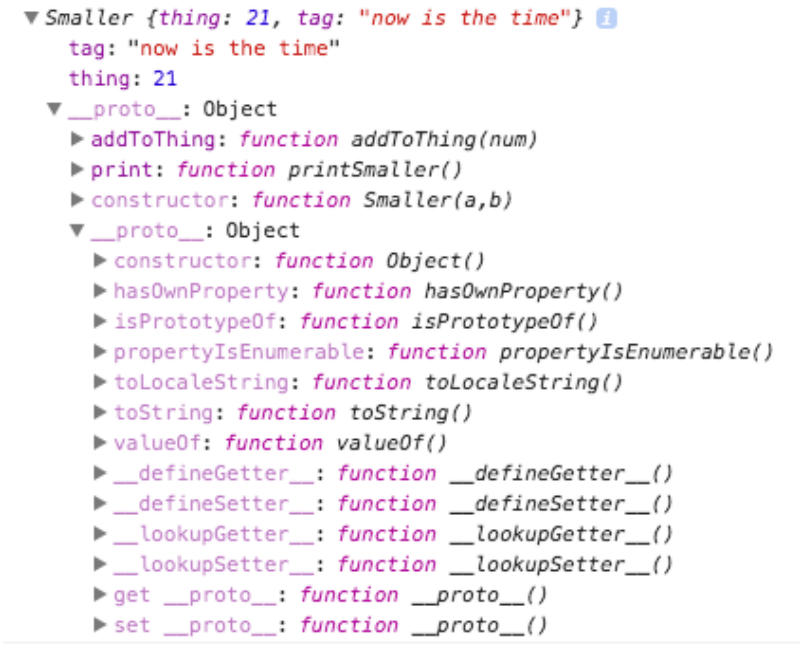
Here's one way to create a class without the *class* expression.

```
function Smaller(a,b){
        console.log("running Smaller");
        this.thing = a;
        this.tag = b;
};
Smaller.prototype.print = printSmaller;
Smaller.prototype.addToThing = addToThing;

function printSmaller(){
        console.log(this.thing, this.tag);
}

function addToThing(num){
        this.thing += num;
        return this.thing;
}

//=================

let small1 = new Smaller(1,"now is the time");
// outputs "running Smaller"

console.log(small1 instanceof Smaller);  // true
small1.print();                          // 1 "now is the time"
small1.addToThing(20);
small1.print();                          // // 21 "now is the time"

small2 = new Smaller(55,"no more time");
small2.print();                          // // 55 "no more time"
```

Note that *small1* created this way behaves exactly the same as *small1* created using the class operator. This is not surprising since, as we will see, the prototypes and the own-properties are the same in the two cases..

```
console.log(Smaller.prototype === Object.getPrototypeOf(small1)); // true
console.log(small1);
```

The output is



the same as in the previous chapter.

Note that the functions *print()* and *addToThing()* are not in the prototype chain of *Smaller*, but rather they are explicitly assigned to be properties of *Smaller.prototype*.

```
console.log(Smaller.print);                    // undefined
console.log(Smaller.addToThing);               // undefined
```

Before creating the subclass *Bigger*, we need to examine the behavior of a very useful function **Object.create()**. Consider first an example.

```
let obj0 = {};
obj0.prototype = {a: 3};
```

Now as discussed previously, *obj0.prototype__proto__* is the universal prototype, created automatically when the object {a:3} is created. Then

```
let obj1 = Object.create(obj0.prototype);
console.log(obj1);
```

The output is

Here's how *Object.create(obj0)* works.

- *obj1* is a new, empty object, where *obj1.__proto__* equals *obj0.prototype*.
- *obj0.prototype* is the object *{a:3}*. This becomes the value of *obj1.__proto__*.
- *obj0.prototype.__proto__* is the universal prototype, was created automatically, and comes along for the ride.

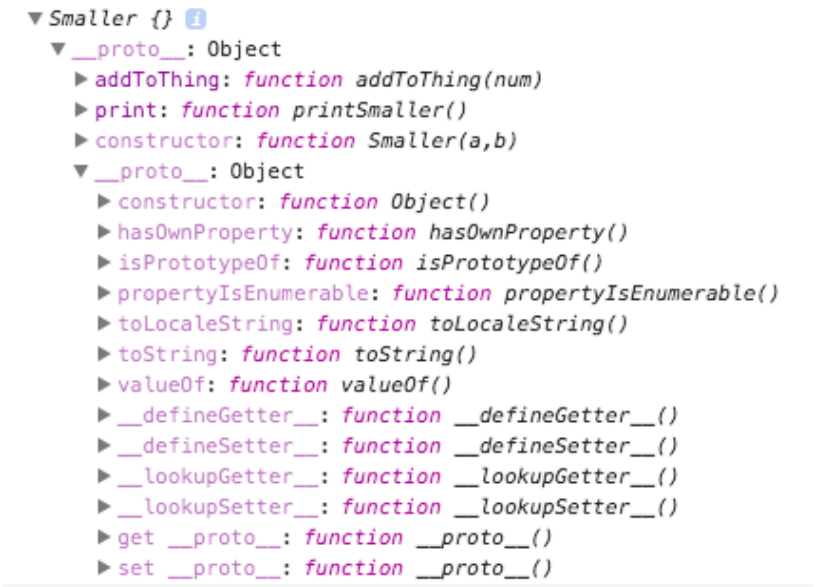
Thus *obj1* has a prototype chain, and *obj1.prototype* is undefined (even though *obj0.prototype* has been defined).

```
console.log(obj1.prototype);          // undefined
```

This procedure is used to allow *Bigger* to inherit from *Smaller*. Consider

```
let proto = Object.create(Smaller.prototype);
console.log(proto);
```

In analogy with the previous example, this is



An empty object, indicated by *Smaller {}*, is created by *Object.create(...),* and it has a *__proto__* hidden pointer which **points to** *Smaller.prototype*. In turn, the *__proto__* pointer (pointing to the universal prototype) of *Smaller.prototype* comes along for the ride. So we are setting up a prototype chain.

subclassing

Here is the subclass.

```
//==================  "class" Bigger (subclass of Smaller) ================

function Bigger(a,b,c){
        Smaller.call(this,a,b); // equivalent to "super()", makes thing, tag
                                // properties of Bigger.
        this.stuff = c;
}

// create an empty object whose prototype is Smaller.prototype
// This object will be Bigger.prototype.

let proto = Object.create(Smaller.prototype);
proto.constructor = Bigger;
proto.print = printBigger;

Bigger.prototype = proto;               // the key statement for inheritance

function printBigger(){
        console.log(this.thing, this.tag, this.stuff);
}
```

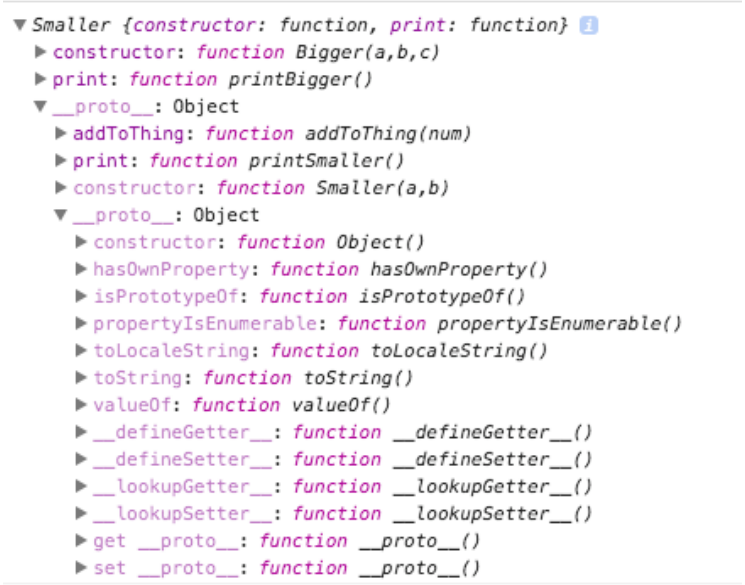
We have created

Bigger.prototype ==> print,constructor,__proto==>Smaller.prototype
Smaller.prototype==> print,addToThing,constructor,__proto__==>universal prototype

We have not modified the prototype chain of Bigger. Therefore, for example, *printBigger* will not be in the prototype chain of *Bigger*, but it will be in the prototype chain of objects created from *Bigger*.

```
console.log(Bigger.prototype);
```

The output is

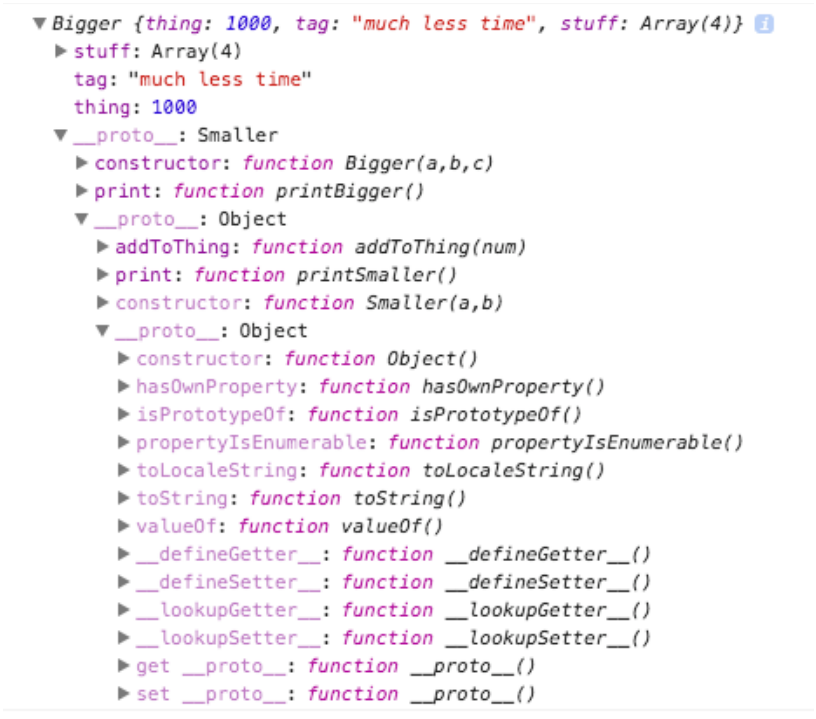


When we do

```
let big1 = new Bigger(1000, "much less time", [1,2,3,4]);
```

Bigger.prototype (shown above) becomes *big1.__proto__*, and we have a prototype chain with inheritance.

```
console.log(big1);
```



This class functions the way we expect for a subclass.

```
let big1 = new Bigger(1000,"much less time", [1,2,3,4]);
big1.print();  //1000 "much less time" (4) [1, 2, 3, 4]-- (invokes printBigger)
console.log(big1 instanceof Bigger);              // true
console.log(big1 instanceof Smaller);             // true

let small1 = new Smaller(1,"now is the time");
console.log(small1 instanceof Bigger);            // false
console.log(small1 instanceof Smaller);           // true
```

To print *Smaller's* print from *big1*, see the end of the previous chapter.

instanceof

The *instanceof* binary operator checks prototypes for identity. That is, it checks whether some prototype in the prototype chain of an object is identical to *aClass.prototype*. (This clearly shows the difference between the prototype chain of an object and *aFuction.prototype*.)

```
function myInstanceOf(object, className) {
        let obj = object;
        while (1){
                obj = Object.getPrototypeOf(obj);
                if(! obj) return false;
                if (obj === className.prototype) return true; // checks pointers
        }
        return false;
}

function Dum(){};

console.log( myInstanceOf(big1,Smaller) );              // true
console.log( myInstanceOf(big1,Bigger) );               // true
console.log( myInstanceOf(big1,Object) );               //' true

console.log( myInstanceOf(big1,Dum) );                  // false
```

It would be a good idea to add to the *while* loop a counter which causes the loop to exit after a given number of iterations. I did not do this here for clarity, but it would be useful in practice.

The function *myInstanceOf* works only because all individual prototype objects are singletons and are pointed to.

classes 3 - lower-level implementation

Here is another implementation of classes. I include it here without much comment. Note function **myNew()**.

```
// funct inherits from superFunct
function myNew(funct, superFunct){
	// add variables in funct to obj
	let obj = {};
	let args = [].slice.call(arguments);
	funct.apply(obj, args.slice(2));

	// now assign obj.__proto__ = funct.prototype
	// this is basically Object.create().
	if(superFunct)
		funct.prototype.__proto__ = superFunct.prototype; // prototype chain
	Object.setPrototypeOf(obj, funct.prototype );
	return obj;
}
// ==============  class Smaller  ====================
function Smaller(a,b){
	this.thing = a;
	this.tag = b;
};
Smaller.prototype.print = printSmaller;
Smaller.prototype.addToThing = addToThing;

function printSmaller(){
	console.log(this.thing, this.tag);
}

function addToThing(num){
	this.thing = this.thing + num;
	return this.thing;
}

//===============  class Bigger ===================
function Bigger(a,b,c){
	Smaller.call(this,a,b);
	this.stuff = c;
}
Bigger.prototype.print = printBigger;

function printBigger(){
	console.log(this.thing, this.tag, this.stuff);
}
```

Objects behave as expected.

```
let small1 = myNew(Smaller, null, 1,"now is the time");
console.log(small1 instanceof Smaller);                    // true

let small2 = myNew(Smaller, null, 177,"no more time");
console.log(small2 instanceof Smaller);                    // true

small1.print();                                            // 1 "now is the time"
small2.print();                                            // 177 "no more time"

small1.addToThing(20);
small2.addToThing(20);

small1.print();                                            // 21 "now is the time"
small2.print();                                            // 197 "no more time"


let big1 = myNew(Bigger, Smaller, 5,"timing",[2,4,6]);

big1.print();                                              // 5 "timing" [2,4,6]

// show that the prototype relationships are correct
console.log(big1 instanceof Bigger);                       // true
console.log(big1 instanceof Smaller);                      // true

console.log(small1 instanceof Smaller);                    // true
console.log(small1 instanceof Bigger);                     // false

console.log(big1);
```

Output of the last expression:

Compare to previous chapters.