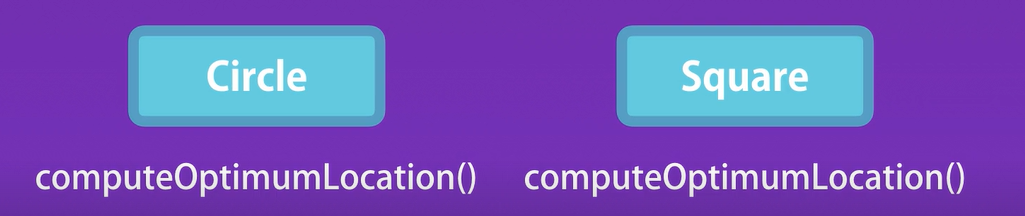


**Inheritance**:

***Inheritance*** is one of the core concepts of the Object Oriented Programming that ***“****enables an object to take on properties and methods of another object***”** and this makes it easy to reuse code in other parts of the application.

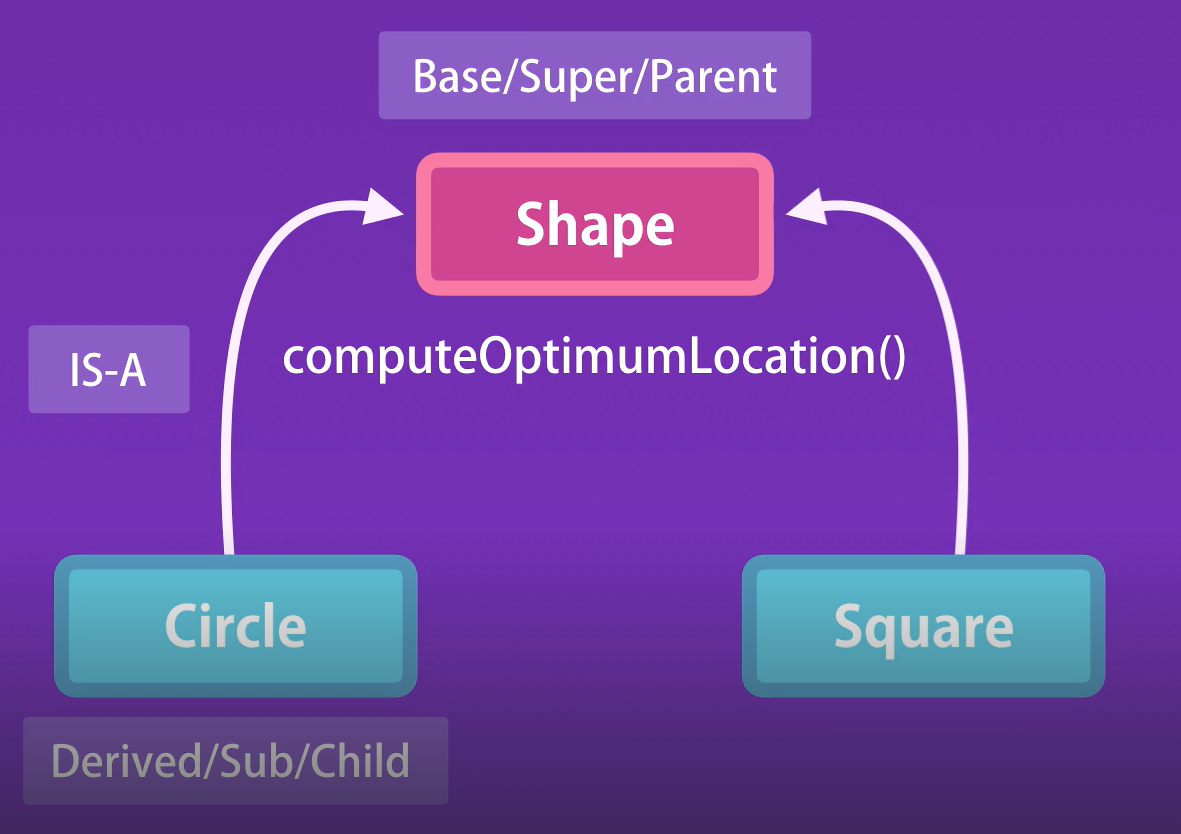
Suppose we have a *Circle* class with one method called *computeOptimumLocation*. Now let us imagine tomorrow we are going to add another class to our application called *Square* which also needs this method.



If the implementation of this method is same across both classes *Circle* and *Square* then we do not need to repeat the same implementation twice because *if there is a bug in the method or some new change then we need to fix it in multiple places*.

This is where *Inheritance* comes to the rescue.

We can define a new class called *Shape*, put our *computeOptimumLocation* method there and have *Circle* and *Square* inherit this method from the Shape class.



In this diagram,

🡪we refer to this *Shape* class as the Base / Super / Parent class.

🡪*Circle* is the Derived / Sub / Child class.

🡪 We refer to inheritance relationship as IS – A relationship. So we say *Circle IS – A* *Shape*.

This is the ***CLASS****ical* definition of inheritance but in case of JavaScript objects we have ***Prototypical*** inheritance.



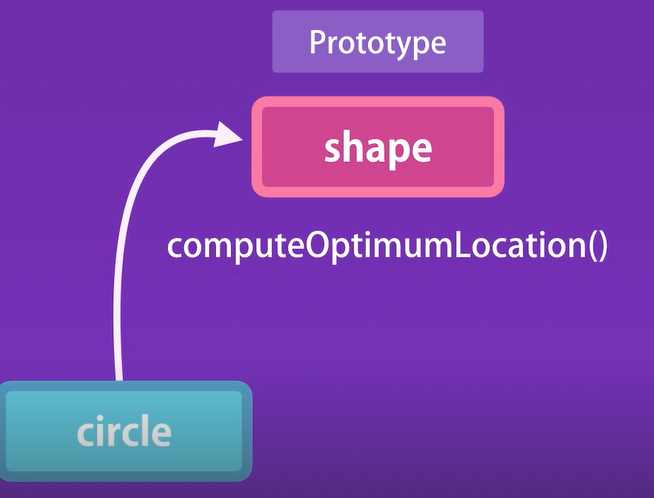
**Prototypes and Prototypical Inheritance**:

The question now is that how can we implement inheritance in objects.

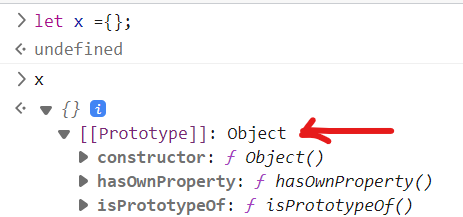
Imagine instead of *Circle* class, we have a *circle* object, we can define another object called *shape* and add all the common behavior or methods in this object.

Then somehow we can link *circle* object to the *shape* object.

The *shape* object will be referred to as the ***prototype*** of the circle. So *prototype is essentially a parent of another object*.



Every object in JavaScript *except only a single object* has a *prototype or parent* and it inherits all the members we find in its prototype.

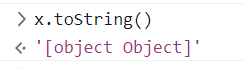


Here on the console we defined an empty object and when we inspect it, we see a property called *Prototype*.

This is the prototype or parent for this x = {} object that we created earlier.

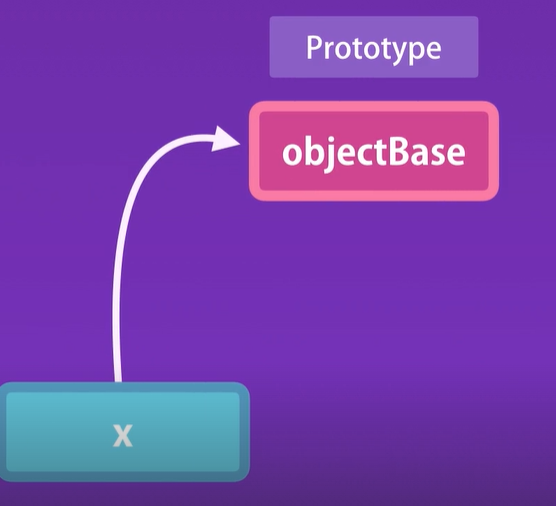
We have related properties and methods like *constructor*. *Every object has a constructor method which references the function that was used to create or construct that object*.

Let us use one of the methods called *toString*,



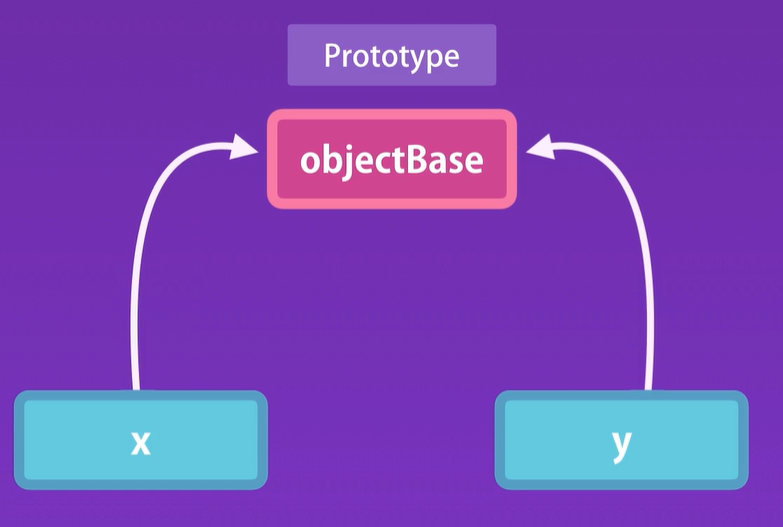
Which gives the default representation of our object.

So we have this *x = {}* object in memory and this object has a link to another object which is its prototype. For our discussion, we call this object as *objectBase* (*not an official term*).



So *every object we create in JavaScript directly or indirectly inherits from objectBase. It is the root of all objects in JavaScript and it does not have a prototype or parent*.

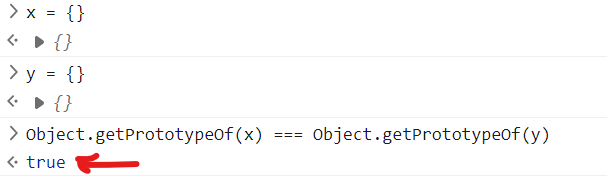
Similarly, we can create another object called y = {}, which refer to the same objectBase as x.



So we have a single instance of objectBase in memory. We can prove it.

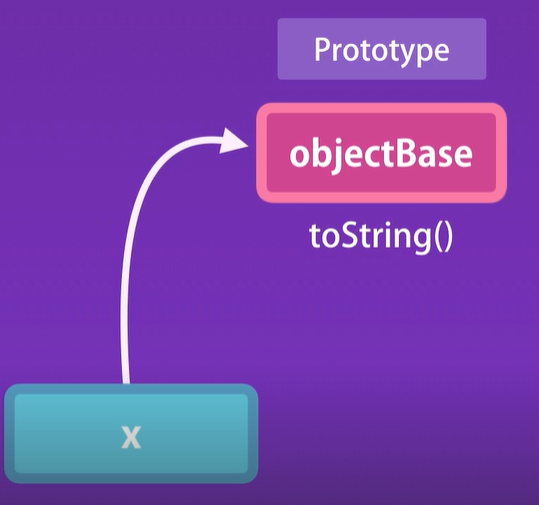
To get the prototype of an object, we have a method called

Object.getPrototypeOf()



Both x and y objects have the exact same prototype.

*Where prototypical inheritance comes into picture*?



We used this *toString* method earlier but how did we access it from x = {} which was an empty object.

When we access a property or a method on an object, JavaScript engine first looks for that property inside the object itself and if it cannot find it then it looks at the prototype of that object.

Again if it cannot even find it in prototype of that object, it will look all the way up to root object which we call objectBase.

***“****This is prototypical inheritance in action when accessing a property or a method of an object, JavaScript engine walks up the prototype chain to find the target member***”**.

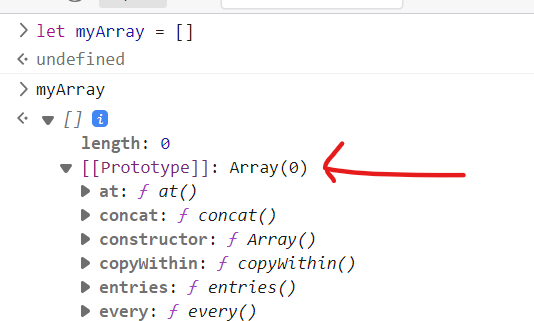
Once again,



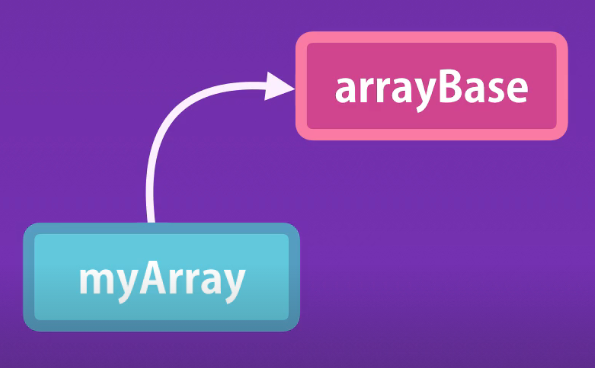
Every object has a prototype or parent except the root object.

**Multi – level Inheritance**:

Here I will define an empty array and inspect it.

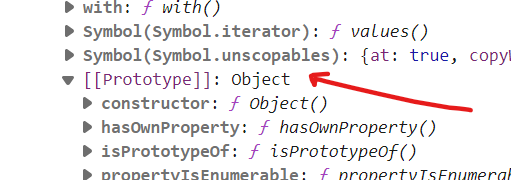


Notice the prototype object or parent of this array. In this object we have all these methods.

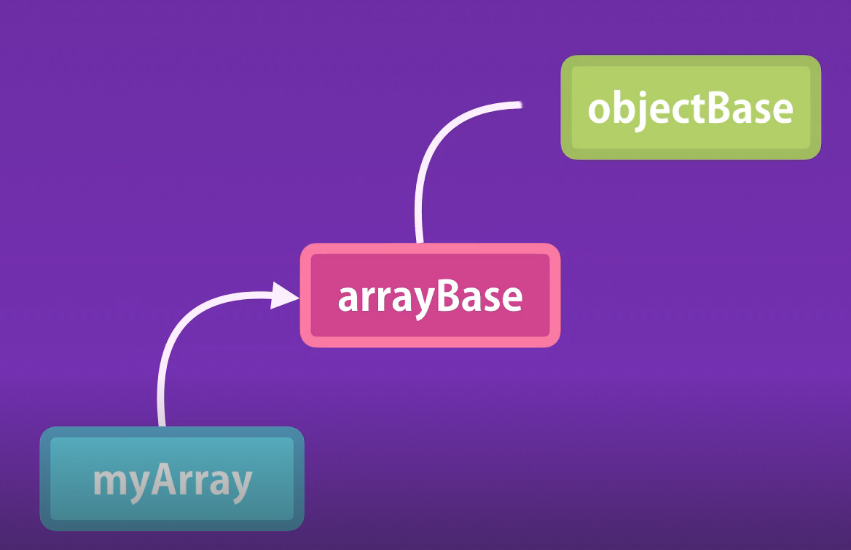


This is what we have in memory, we have *myArray* which derives from let us say *arrayBase*.

Let us inspect the prototype of arrayBase object.



We have objectBase or root object here.

This is the visualization of what we have in memory.

Here, myArray 🡪 arrayBase 🡪 objectBase

This is what we call *multilevel inheritance*.

What about the *objects we create using the custom constructors*?

function Circle(radius) {

  this.radius = radius;

  this.draw = function () {

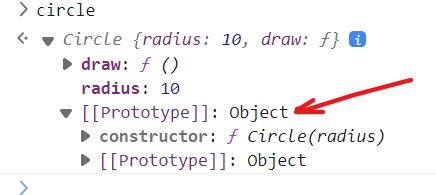
    console.log("Draw");

  };

}

const circle = new Circle(10);

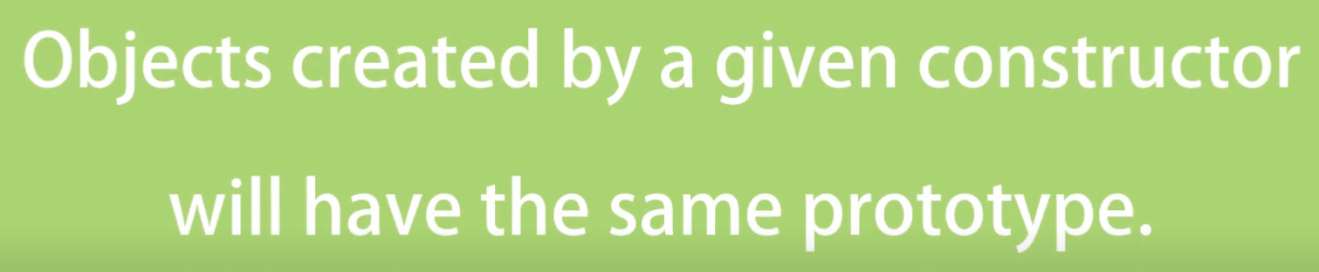
Here we are inspecting circle object,



This prototype is the prototype for all *circle* objects that we create using our *Circle* constructor, we will call it *circleBase*.

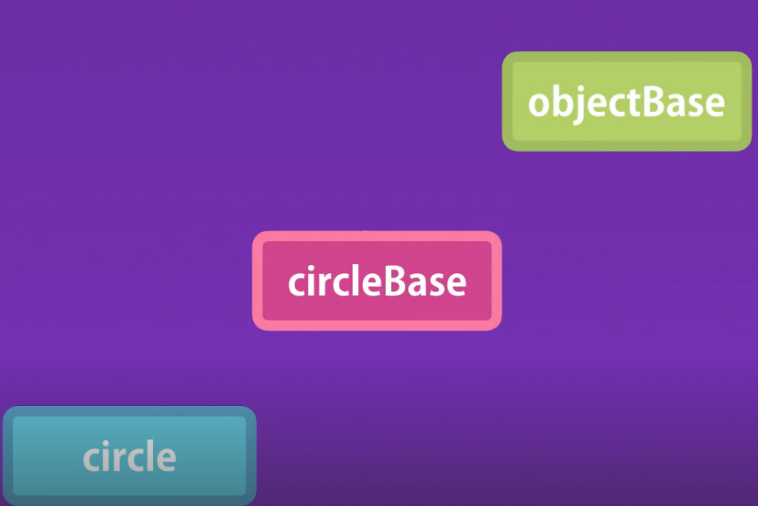
So every time we call the Circle constructor to create a new circle object, the constructor will create a new object and set its prototype property to circleBase.

*In other words*,



So *all circle objects created by Circle constructor will have the same prototype* and *similarly all arrays created by the array constructor will have the same prototype*.

This is what we have in memory,



Every circle object inherits from circleBase which further inherits from objectBase.

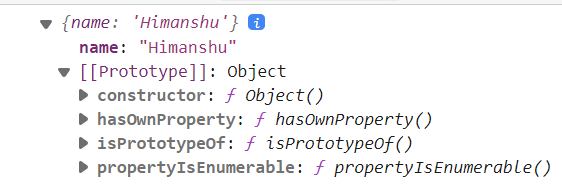
**Property descriptors**:

Here we define a new object,

let person  = {name: "Himanshu"}

console.log(person);

Here we have person object in console with all methods and properties available in its prototype.



However if we try to iterate over the members of this person object we are not going to see any properties or methods of this object.

let person = { name: "Himanshu" };

for (let key in person) {

  console.log(key); //name

}

console.log(Object.keys(person)); //[ 'name' ]

Only ‘*name’* property is here.

*Why we cannot iterate over properties and methods defined in the objectBase*?

Reason is because in JavaScript, *our properties have attributes attached to them. Sometimes these attributes prevent a property from being enumerated*.

To see attributes in action,

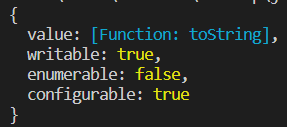
First get prototype of the person (objectBase) and let us see attributes attached to one of the method called *toString* (). For this we use *Object.****getOwnPropertyDescriptor*** which will provide us with property descriptor.

let person = { name: "Himanshu" };

let objectBase = Object.getPrototypeOf(person);

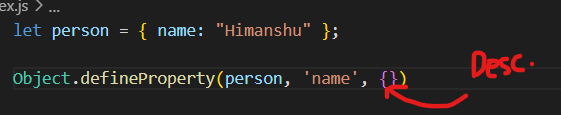
let descriptor = Object.getOwnPropertyDescriptor(objectBase, "toString");

console.log(descriptor);

🡨*Our toString property descriptor*.

1. Here, configurable: true (*means we can delete this member if we want to*).
2. enumerable: false (*we cannot iterate over it*)
3. Writable: true (*we can overwrite this method or change its implementation or set its value*).

Similarly *we can also set these attributes for the members of our custom objects* using *Object.****defineProperty*** like this.

<

First argument is the name of the object, second argument is the name of the key and *third argument is the property descriptor object*, where we add attributes of this object.

To *make it a read only property*,

let person = { name: "Himanshu" };

Object.defineProperty(person, "name", {

  writable: false, //set to false

});

person.name = "john"; // writing a new value to property

console.log(person.name); //Himanshu

Name is still not changed.

*To make it non – iterable*,

let person = { name: "Himanshu" };

Object.defineProperty(person, "name", {

  enumerable: false,

});

console.log(Object.keys(person)); // []

Now *name* key is not visible.

To *make this property unable to delete*:

let person = { name: "Himanshu" };

Object.defineProperty(person, "name", {

  configurable: false,

});

delete person.name; //We delete this property

console.log(person); // { name: 'Himanshu' }

Our *name* property is still here.

**Constructor Prototypes**:

We have learned that every object in JavaScript except the root object has a prototype or a parent.

The proper way to get the prototype of an object is by using Object.getPrototypeOf.

Object.getPrototypeOf(myObj)

It is same as what we saw in console as,

myObj.\_\_proto\_\_

What we get as the result is the prototype or the parent of *myObj*.

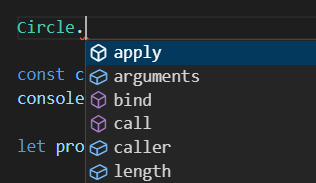
Now one thing we need to know is that *constructors also have a prototype property*.

function Circle(radius) {

  this.radius = radius;

}

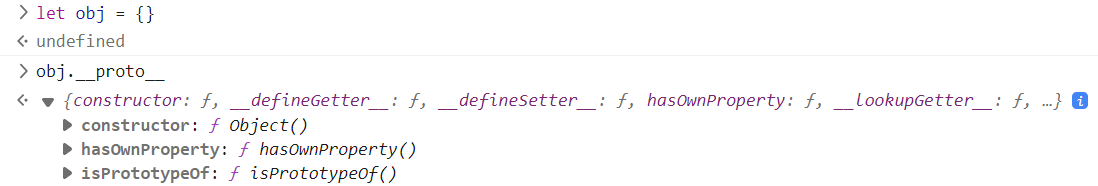
Here we have a *Circle* constructor function and we know that in JavaScript functions are objects, so they have property and methods.

🡨 We can see these properties and methods available to every function.

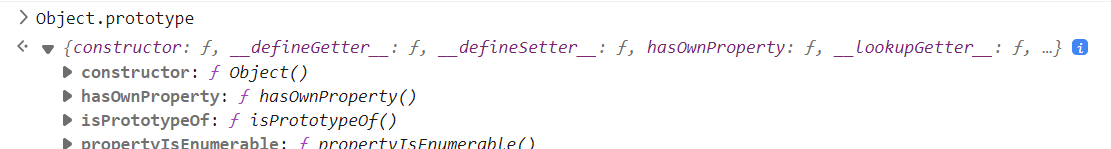
One of the property is *prototype*. *This is the object that will be used as the parent for objects created by the Circle constructor*.

Let us see this in action,

I will define a simple object here using object literal syntax and then check its \_\_*proto*\_\_ property (*to see its parent or objectBase*).

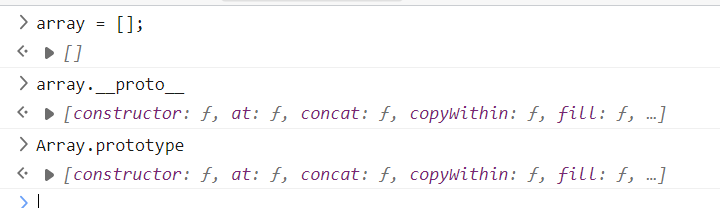


You know that this *obj* object is created using *Object* constructor function which also has a *prototype* property *which is gives us the prototype or parent of all objects created by this constructor*.

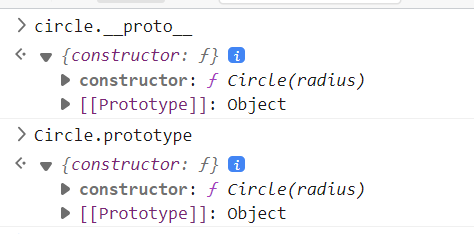


So we see that result from obj.\_\_proto\_\_ and Object.prototype are equal, the parent of obj and prototype for its constructor.

Same goes for arrays initialized using array literal syntax.



And our *circle* object and *Circle* constructor function.



**Prototypes vs Instance members**:

Consider this example,

function Circle(radius) {

  this.radius = radius;

  this.draw = function () {

    console.log("draw");

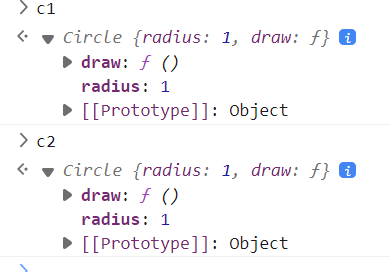
  };

}

const c1 = new Circle(1);

const c2 = new Circle(1);

Log these objects on console,



With this current implementation, if we had 1000 circle objects in memory, we are going to have 1000 copies of the draw method as well.

Now this is a very simplified example, in a real world application, your objects might have several methods.

So if we want to have large number of objects in the memory, we will waste lot of memory space by keeping copies of all these methods.

*So what is the solution*?

We know how *prototypical inheritance* works. *When we access a property or a method on an object, JavaScript engine first looks at the object itself and if it cannot find that property or method, it will look at the prototype of that object*.

So the solution is that we take out the *draw* method out of this circle object and put it in its prototype. We are going to have a single instance of this prototype in the memory which we call circleBase (*so a single instance of draw method*).

Circle constructor is an object and since JavaScript objects are dynamic we can add a member to its prototype like this.

function Circle(radius) {

  this.radius = radius;

}

Circle.prototype.draw = function () {

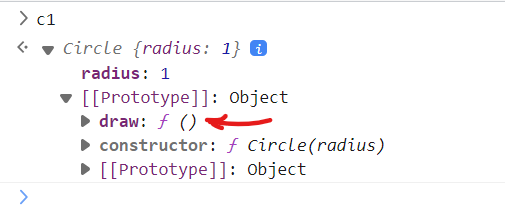
  console.log("draw");

};

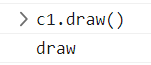
const c1 = new Circle(1);

const c2 = new Circle(1);

Now*, draw* method is now inside circleBase.



Due to prototypical inheritance we can still access it,



So basically we have two kinds of properties and methods in JavaScript,

* Instance members
* Prototype members

*How can we override objectBase methods*?

Take example of *toString* method.

🡨By default it returns this string here.

We can override the implementation of this method in the prototype of our circle object.

Circle.prototype.toString = function () {

  return "circle with radius " + this.radius;

};



Note: Even though we have another implementation of toString method but our new implementation will be used because this is more accessible.

Note: In both instance and prototype members we can reference other members.

For example in draw method, which is a prototype method we can easily call an instance method.

function Circle(radius) {

  this.radius = radius;

  this.move = function () { 🡪 Instance member

    console.log("move");

  };

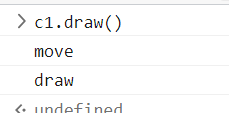
}

Circle.prototype.draw = function () {

  this.move(); 🡪 called inside prototype

  console.log("draw");

};



We get move and draw both.

By the same token in an instance member we can reference a prototype member.

function Circle(radius) {

  this.radius = radius;

  this.move = function () {

    this.draw(); 🡪 prototype member

    console.log("move");

  };

}

Circle.prototype.draw = function () {

  console.log("draw");

};

**Iterating Instance and Prototype members**:

In this example,

function Circle(radius) {

  //Instance members

  this.radius = radius;

  this.move = function () {

    console.log("move");

  };

}

//Prototype members

Circle.prototype.draw = function () {

  console.log("draw");

};

const c1 = new Circle(1);

We have circle object with two instance members (*radius and move*) and one prototype member (*draw*).

*It does not matter where you change the prototype*. So in this case we are modifying the prototype before creating an object. We can also do the opposite.

const c1 = new Circle(1);

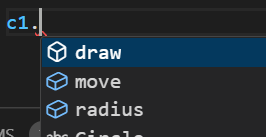
//Prototype members

Circle.prototype.draw = function () {

  console.log("draw");

};

Create an object first and then modify the prototype.

🡨, *draw* method is still available on the Circle object.

Because here we are dealing with object references. So we have a single object in memory and as soon as we modify that, all the changes are immediately visible.

In this lecture, we are going to learn how to iterate over instance vs prototype properties.

*Iterating instance members*:

Use Object.keys () method to get only the instance members.

const c1 = new Circle(1);

console.log(Object.keys(c1)); [ 'radius', 'move' ]

*Iterating all members*:

Use for – in loop

const c1 = new Circle(1);

for (let key in c1) console.log(key); radius move draw

It returns all the members (*instance + prototype*)

Note: In JavaScript we often use the word *own* instead of instance to describe instance members. So in some documents / tutorials we may hear *own properties* Vs *prototype property*.

It is a good way to remember this useful function called *hasOwnProperty* to determine whether a given member is an instance member or not.

console.log(c1.hasOwnProperty("move")); //true

console.log(c1.hasOwnProperty("draw")); //false

**Avoid Extending the built in Objects**:

So we have learned how easy it is to modify the prototype of an object.

So we might be tempted to do something like this.

Array.prototype.shuffle = function(){

  //...

}

We have added a new method to our built in Array constructor and call it from an *array* object.

const array = []

array.shuffle()

While this is very easy to accomplish in JavaScript, we should avoid it and not modify built in objects in JavaScript.

The reason behind is that tomorrow *it might be possible that we use a 3rd party library and in that library someone also has extended the array prototype and added the shuffle method but with a different implementation*.

*The simple rule is*,

**Do not modify objects you do not own**.

Yes, JavaScript is a dynamic language which makes it really easy to add properties and methods to an existing object, but that does not mean that you should modify the built in objects.