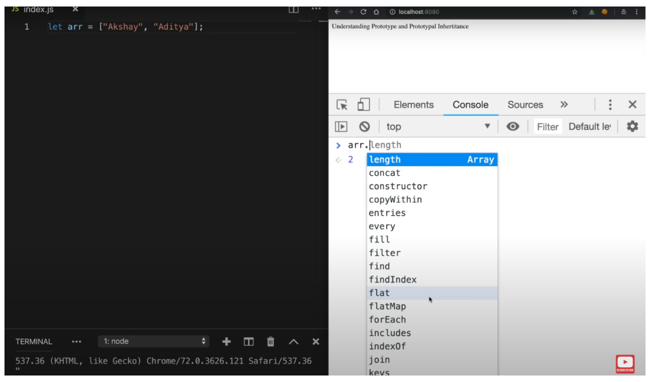
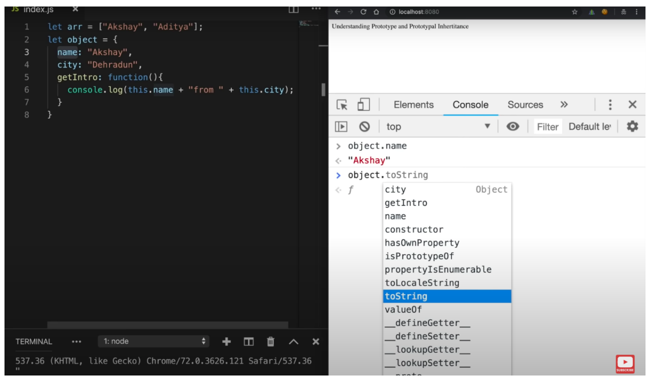
Prototype and Prototypal Inheritance in JavaScript

Thursday, 24 August 2023   7:58 PM



```
let arr = ["Akshay", "Aditya"];
```

```
> arr.length
< 2
```



```
let arr = ["Akshay", "Aditya"];
let object = {
  name: "Akshay",
  city: "Dehradun",
  getIntro: function(){
    console.log(this.name + "from " + this.city);
  }
}
```

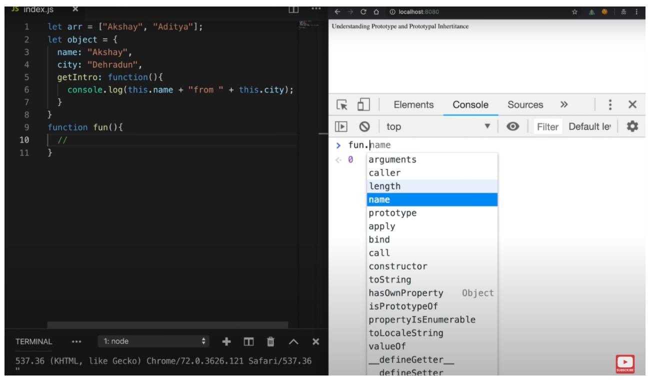
```
> object.name
< "Akshay"
> object.toString
< f
```

→ How are arrays & objects getting access to these inbuilt f^n's?

→ these are where prototypes come to picture.

Whenever we create a JS object, JS engine automatically attaches the object with some hidden properties & f^n's which can be accessed using .(dot) operator

→ Even f^n's have access to some hidden properties like call(), apply() & bind().



```
let arr = ["Akshay", "Aditya"];
let object = {
  name: "Akshay",
  city: "Dehradun",
  getIntro: function(){
    console.log(this.name + "from " + this.city);
  }
}
function fun(){
  //
}
```

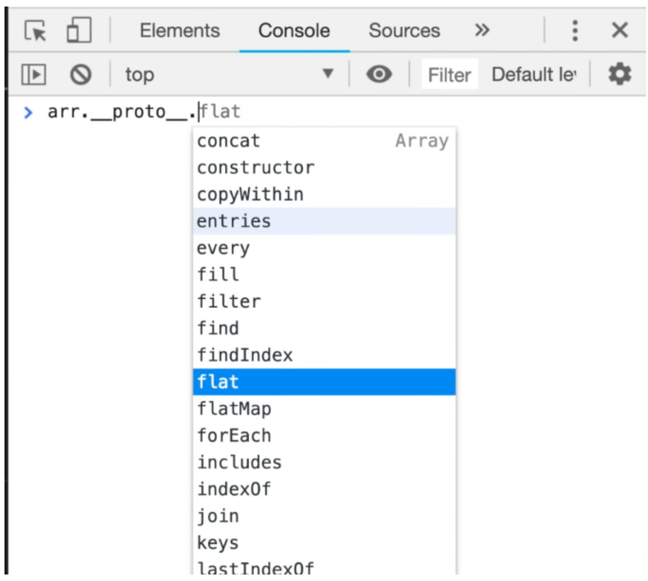
```
> fun.name
< 0
```

→ Even variables get access to some hidden f^n's & properties.

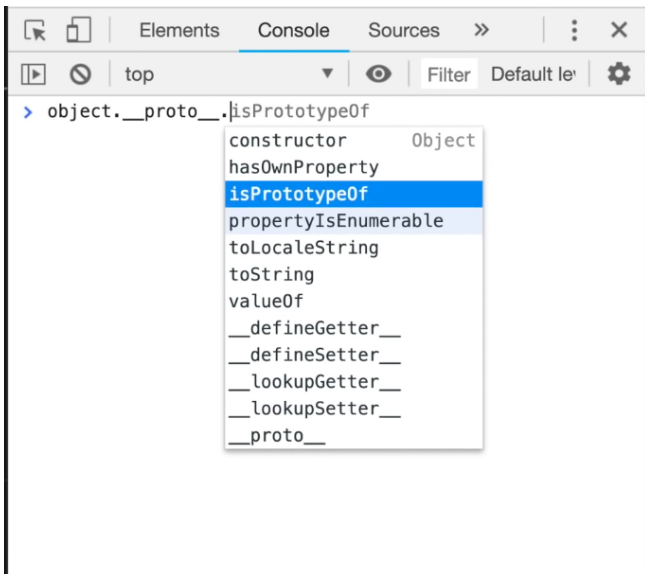
→ This is called prototype.

→ whenever an object is created, JS engine automatically puts these hidden properties inside an object & attaches it to your object.
This is how we get access to those properties & methods.

→ __proto__ is the object where JS is putting all these hidden f^n's which can be accessed using .(dot) operator.

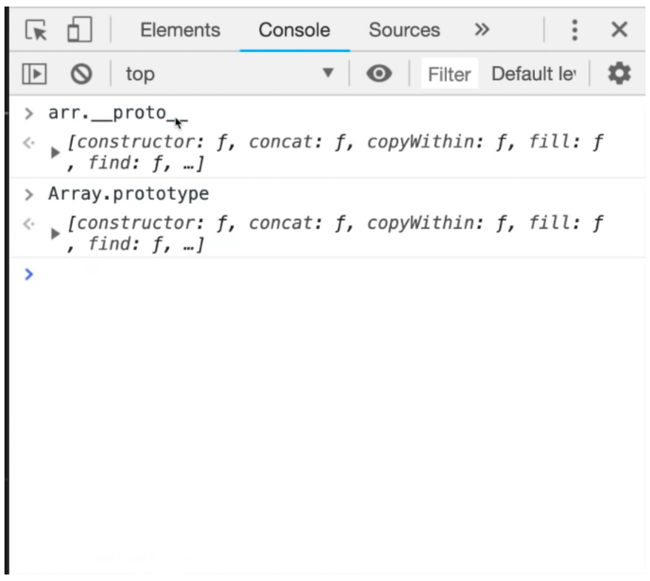


```
> arr.__proto__.flat
```



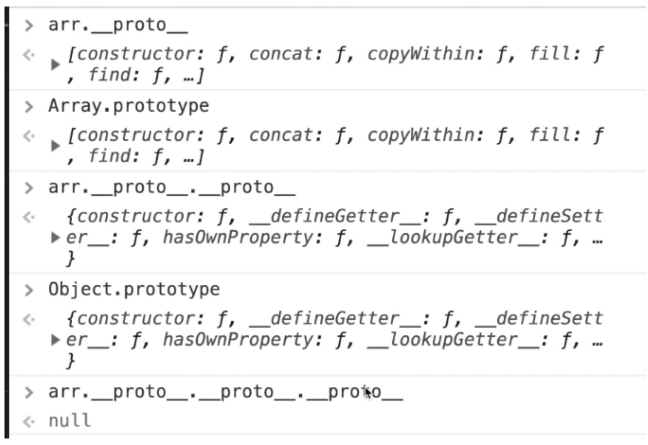
```
> object.__proto__.isPrototypeOf
```

→ This __proto__ object is attached to our object.



```
> arr.__proto__
<· ▸[constructor: f, concat: f, copyWithin: f, fill: f
   , find: f, …]
> Array.prototype
<· ▸[constructor: f, concat: f, copyWithin: f, fill: f
   , find: f, …]
>
```

→ arr.__proto__ is same as Array.prototype.



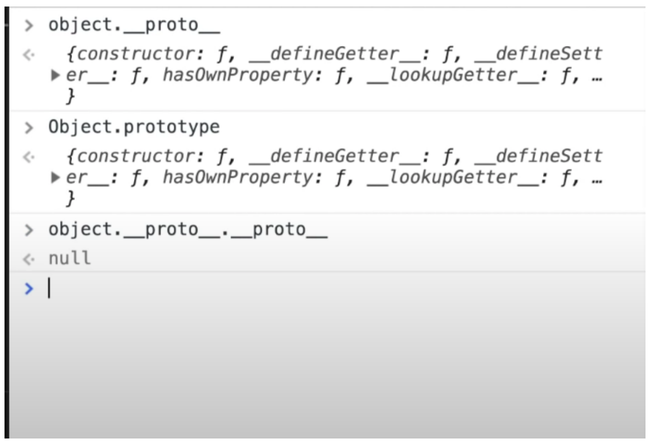
```
> arr.__proto__
<· ▸[constructor: f, concat: f, copyWithin: f, fill: f
   , find: f, …]
> Array.prototype
<· ▸[constructor: f, concat: f, copyWithin: f, fill: f
   , find: f, …]
> arr.__proto__.__proto__
<·  {constructor: f, __defineGetter__: f, __defineSett
  ▸er__: f, hasOwnProperty: f, __lookupGetter__: f, …
   }
> Object.prototype
<·  {constructor: f, __defineGetter__: f, __defineSett
  ▸er__: f, hasOwnProperty: f, __lookupGetter__: f, …
   }
> arr.__proto__.__proto__.__proto__
<· null
```

→ Now arr.__proto__ is same as Array.prototype

→ Array.prototype's __proto__ which is arr.__proto__.__proto__ is same as object.prototype.

→ If we find the prototype of object.prototype, it will be arr.__proto__.__proto__.__proto__ which is null.

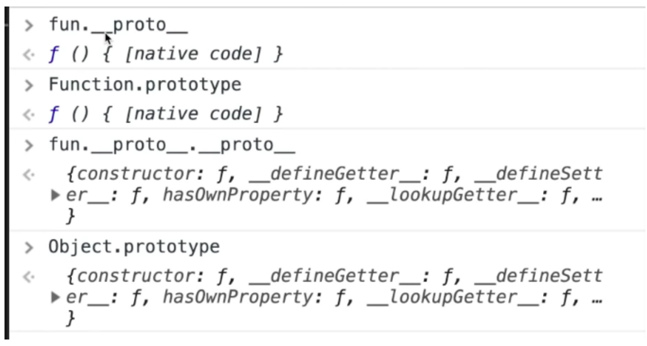
→ This is known as prototype chaining.



```
> object.__proto__
<·  {constructor: f, __defineGetter__: f, __defineSett
  ▸er__: f, hasOwnProperty: f, __lookupGetter__: f, …
   }
> Object.prototype
<·  {constructor: f, __defineGetter__: f, __defineSett
  ▸er__: f, hasOwnProperty: f, __lookupGetter__: f, …
   }
> object.__proto__.__proto__
<· null
> |
```

→ object.__proto__ = Object.prototype

→ So if we find object.__proto__, it will be null.



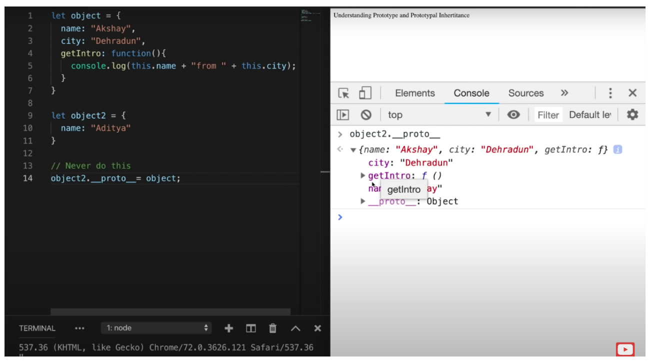
```
> fun.__proto__
<· f () { [native code] }
> Function.prototype
<· f () { [native code] }
> fun.__proto__.__proto__
<·  {constructor: f, __defineGetter__: f, __defineSett
  ▸er__: f, hasOwnProperty: f, __lookupGetter__: f, …
   }
> Object.prototype
<·  {constructor: f, __defineGetter__: f, __defineSett
  ▸er__: f, hasOwnProperty: f, __lookupGetter__: f, …
   }
```

→ In case of f^n's, f^n.__proto__ = function.prototype which is nothing.

→ If we do fun.__proto__.__proto__ we get Object.prototype.

→ So, basically everything in JS is an object.

→ We should never do this, but if we change where object2.__proto__ points then we can get access to that object's variables & methods after doing this.

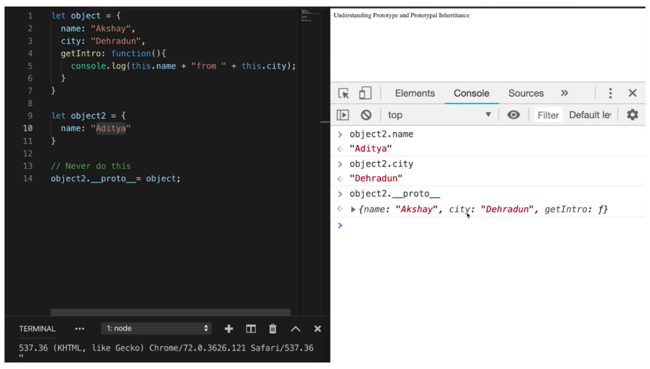


```
let object = {
  name: "Akshay",
  city: "Dehradun",
  getIntro: function(){
    console.log(this.name + "from " + this.city);
  }
}

let object2 = {
  name: "Aditya"
}

// Never do this
object2.__proto__= object;
```

```
537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36
```

```
> object2.__proto__
<· ▾{name: "Akshay", city: "Dehradun", getIntro: f}
    city: "Dehradun"
   ▸getIntro: f ()
    nar getIntro ay"
   ▸__proto__: Object
>
```

→ So, after line 14, if we try to access 'object2.city', so first JS will check the main object (object2) for city, then it will check the __proto__ if it doesn't find city in main object, if there also it is not present then it will go to the __proto__ of __proto__. So that's how it goes through the whole chain.



```
let object = {
  name: "Akshay",
  city: "Dehradun",
  getIntro: function(){
    console.log(this.name + "from " + this.city);
  }
}

let object2 = {
  name: "Aditya"
}

// Never do this
object2.__proto__= object;
```

Understanding Prototype and Prototypal Inhertitance

```
> object2.name
<- "Aditya"
> object2.city
<- "Dehradun"
> object2.__proto__
<- ▸ {name: "Akshay", city: "Dehradun", getIntro: f}
>
```

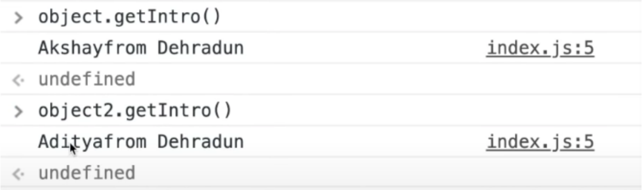
```
537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36
"
```

→ So this is how object2 inherits from object & this is what prototypal inheritance is.

→ Thus, object2 is 'inheriting' properties from object.

→ Can we access fn using inheritance?

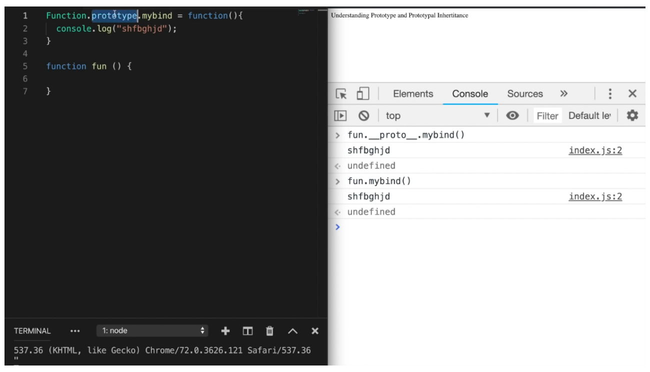
ans. Yes, we can.



```
> object.getIntro()
  Akshayfrom Dehradun                      index.js:5
<- undefined
> object2.getIntro()
  Adityafrom Dehradun                      index.js:5
<- undefined
```

→ So in 2nd case, 'this' keyword points to object2 & gets 'name' from object2. But when it does not find 'city' in object2, it goes through the prototype chain and access 'city' from object.

Note: Now, when we use Function.prototype we set the mybind() fn to Function.prototype & this fn can then be accessed by all fn's by using .(dot) operator.



```
Function.prototype.mybind = function(){
  console.log("shfbghjd");
}

function fun () {

}
```

Understanding Prototype and Prototypal Inhertitance

```
> fun.__proto__.mybind()
  shfbghjd                                 index.js:2
<- undefined
> fun.mybind()
  shfbghjd                                 index.js:2
<- undefined
>
```

```
537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36
"
```