Object Oriented JavaScript & Prototype Chaining

Narendra Sisodiya

@nsisodiya

Narendra Sisodiya

I am JavaScript Developer

I develop Large Scale, Scalable, Single Page Applications

@nsisodiya

github.com/nsisodiya

speakerdeck.com/nsisodiya

Agenda

- Understand **Object**
- Understand **this** keyword
- Understand **Object.create**
- Understand **Object.prototype**
- Understand **Prototype chaining**
- Understand **Prototype Inheritance**
  - with **this** and **new** keyword
  - without **this** and **new** keyword

Objects

- var o = {};

  // is equivalent to:
- var o = Object.create(Object.prototype);

An Object is dynamic collection of properties

- Every properties is key-value pair
- key is traditionally string, but with WeakMap, it can be any other object.

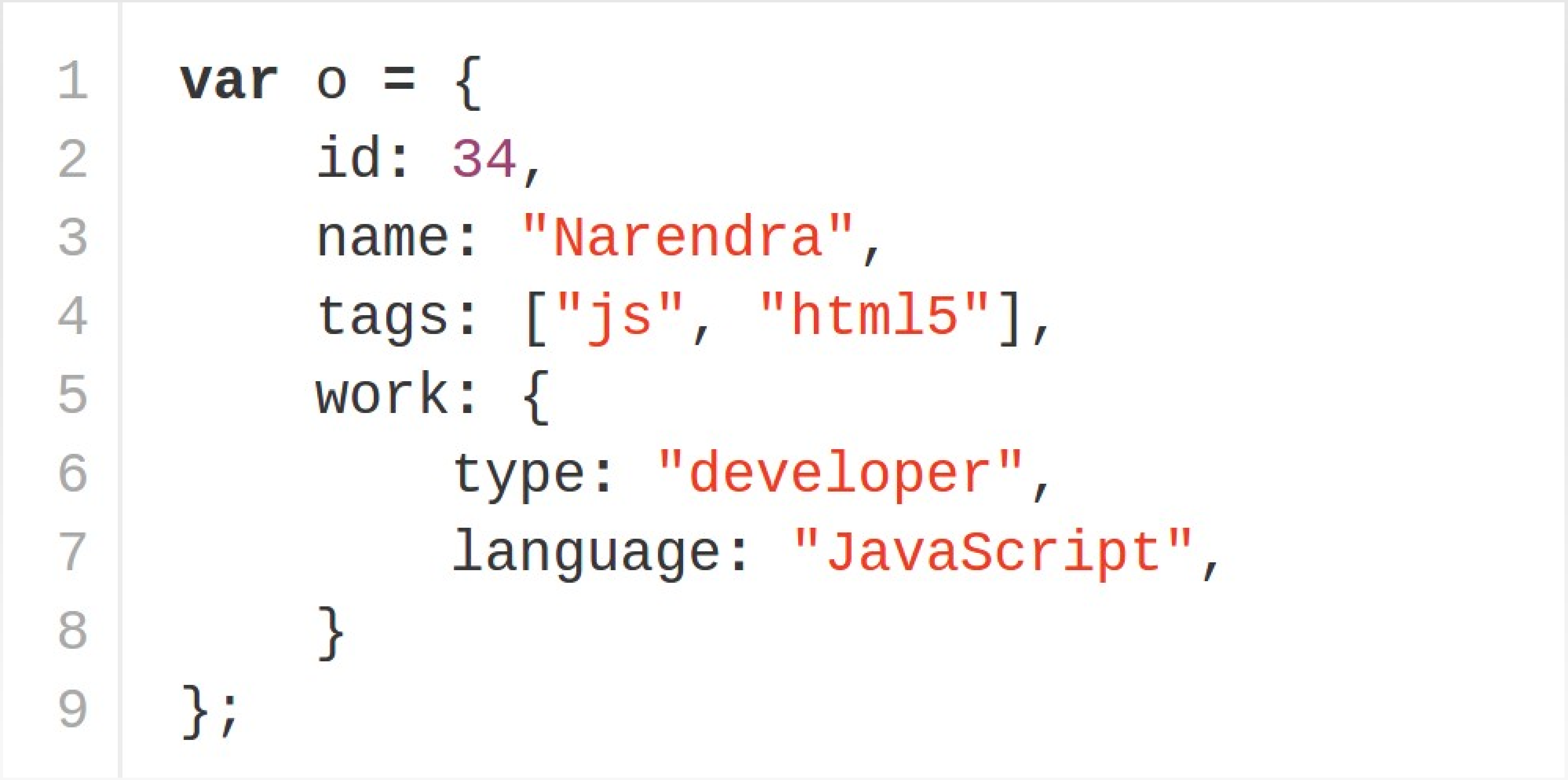
Getter/Setter/Delete

- var o = {};
- Set
  - o.id = 34
  - o.name = "Narendra"
- Get
  - console.log(o.id)
  - console.log(o.name)
- Delete
  - delete o.id
  - delete o.name

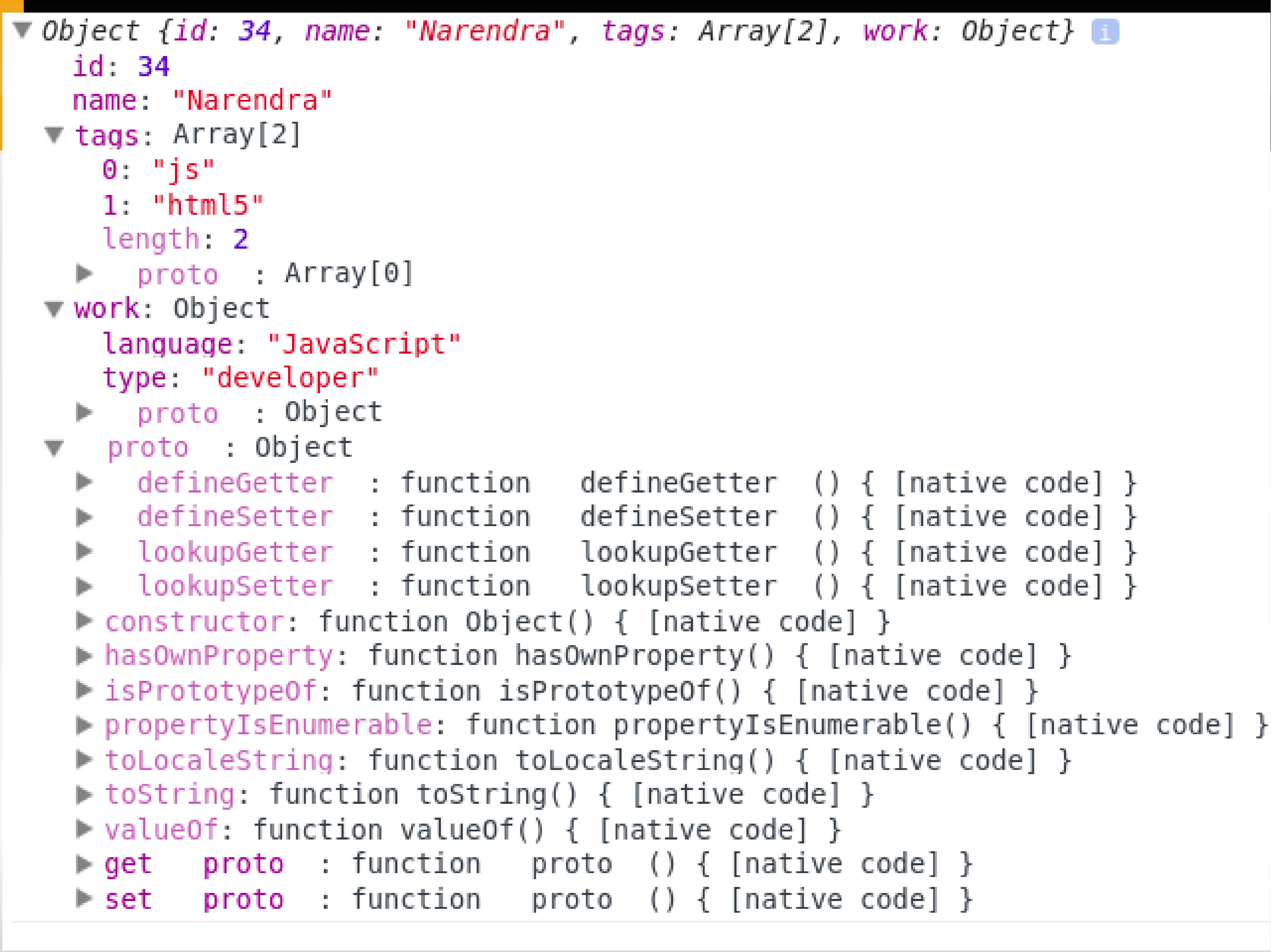
- o["name"] = "Narendra"

Dot Notation
and
Bracket Notation
are exactly same

Object Literals



```javascript
var o = {
    id: 34,
    name: "Narendra",
    tags: ["js", "html5"],
    work: {
        type: "developer",
        language: "JavaScript",
    }
};
```



```
▼ Object {id: 34, name: "Narendra", tags: Array[2], work: Object} i
    id: 34
    name: "Narendra"
  ▼ tags: Array[2]
      0: "js"
      1: "html5"
      length: 2
    ▶   proto  : Array[0]
  ▼ work: Object
      language: "JavaScript"
      type: "developer"
    ▶   proto  : Object
  ▼   proto  : Object
    ▶   defineGetter  : function   defineGetter  () { [native code] }
    ▶   defineSetter  : function   defineSetter  () { [native code] }
    ▶   lookupGetter  : function   lookupGetter  () { [native code] }
    ▶   lookupSetter  : function   lookupSetter  () { [native code] }
    ▶ constructor: function Object() { [native code] }
    ▶ hasOwnProperty: function hasOwnProperty() { [native code] }
    ▶ isPrototypeOf: function isPrototypeOf() { [native code] }
    ▶ propertyIsEnumerable: function propertyIsEnumerable() { [native code] }
    ▶ toLocaleString: function toLocaleString() { [native code] }
    ▶ toString: function toString() { [native code] }
    ▶ valueOf: function valueOf() { [native code] }
    ▶ get   proto  : function   proto  () { [native code] }
    ▶ set   proto  : function   proto  () { [native code] }
```

WHAT IF I TOLD YOU

JavaScript do not have CLASSES

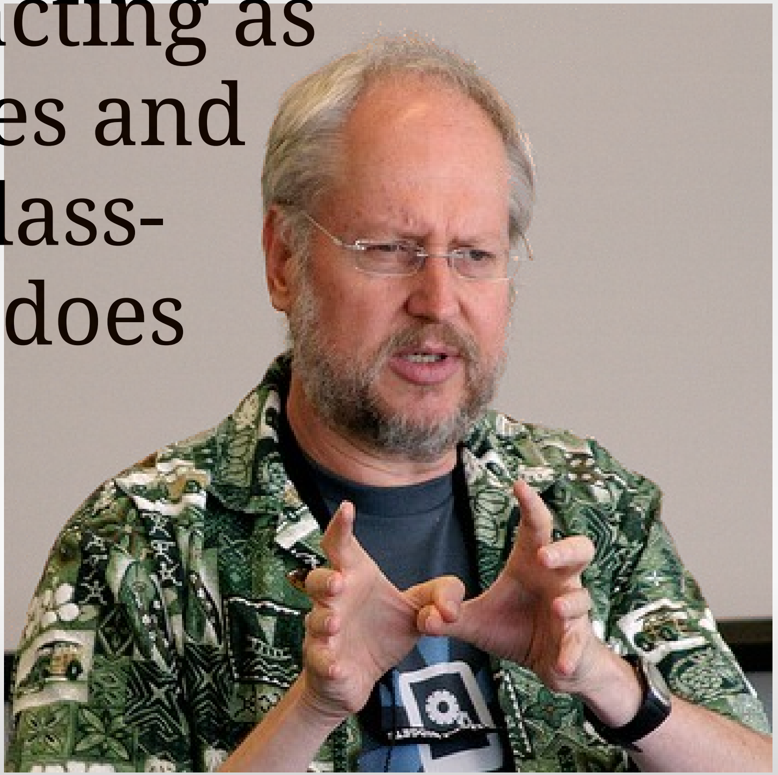
Surprise : There are no CLASSES

In JavaScript

An Object
can **extend**
from
Another Object

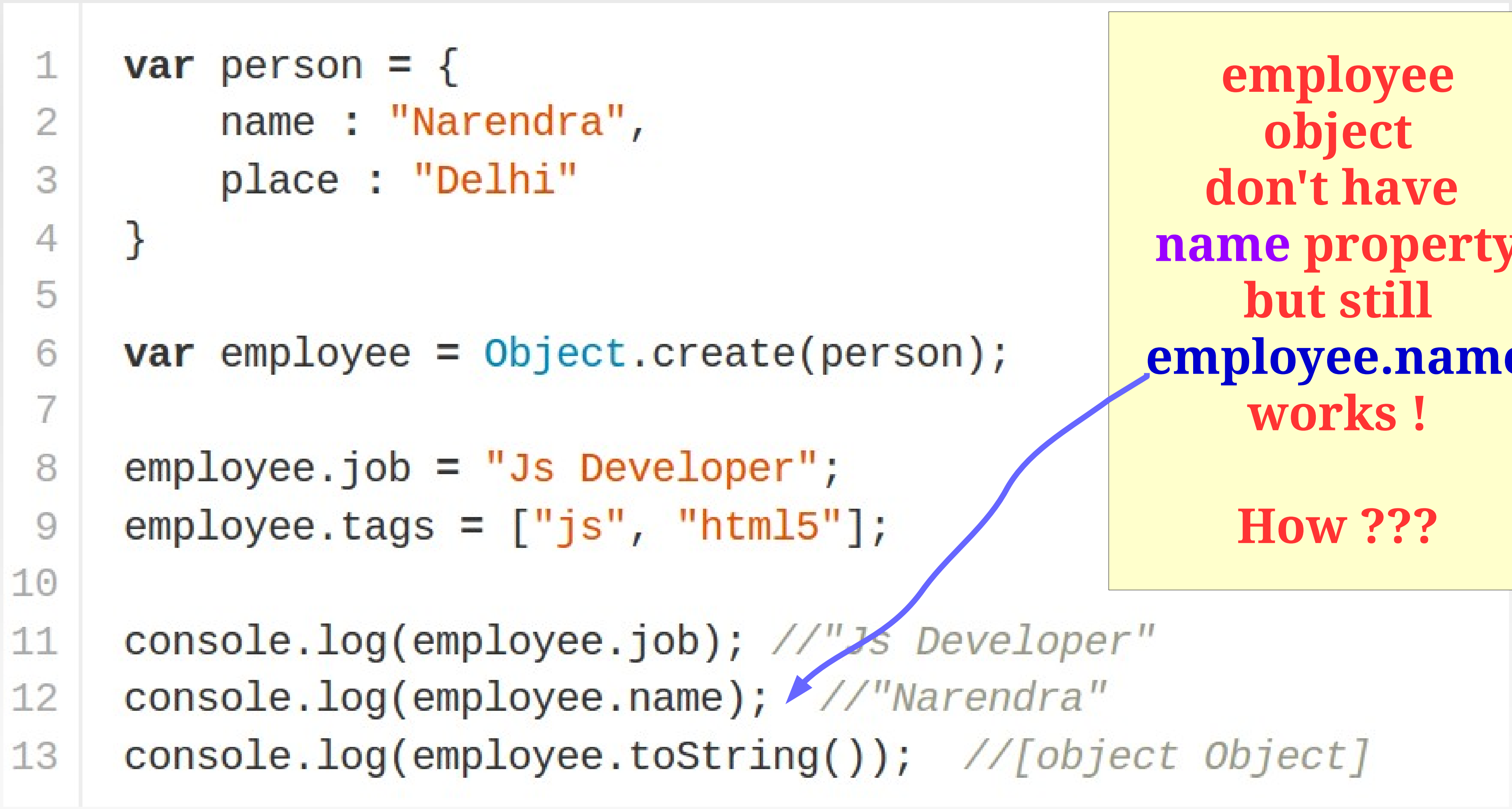
Is JavaScript object-oriented?

- It has objects which can contain data and methods that act upon that data. Objects can contain other objects. It does not have classes, but it does have constructors which do what classes do, including acting as containers for class variables and methods. It does not have class-oriented inheritance, but it does have prototype-oriented inheritance.
  - By - Douglas Crockford



How ???

using Object.create

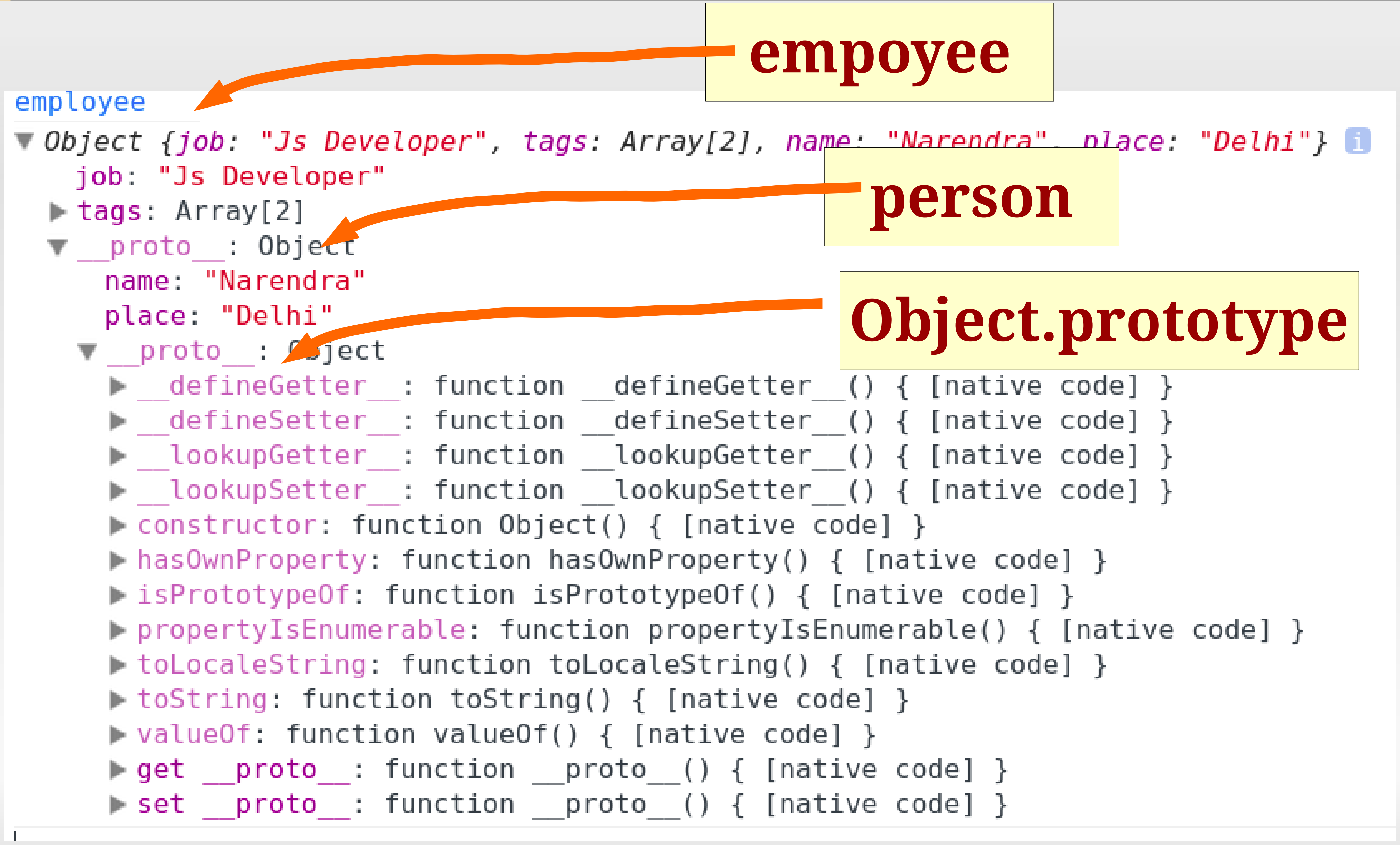


```
var person = {
    name : "Narendra",
    place : "Delhi"
}

var employee = Object.create(person);

employee.job = "Js Developer";
employee.tags = ["js", "html5"];

console.log(employee.job); //"Js Developer"
console.log(employee.name); //"Narendra"
console.log(employee.toString());  //[object Object]
```

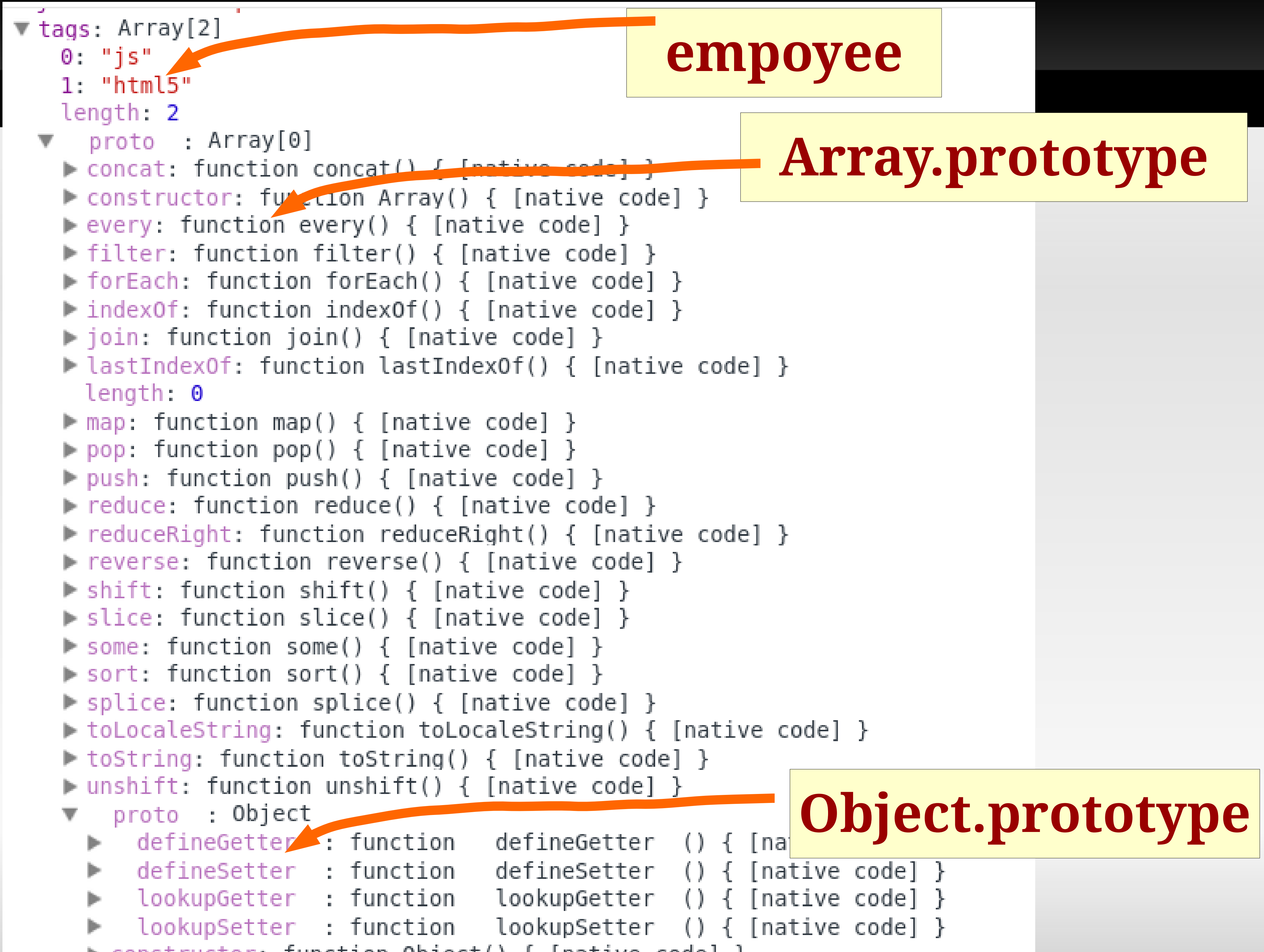
employee object don't have name property but still employee.name works !

How ???

Object extending Object

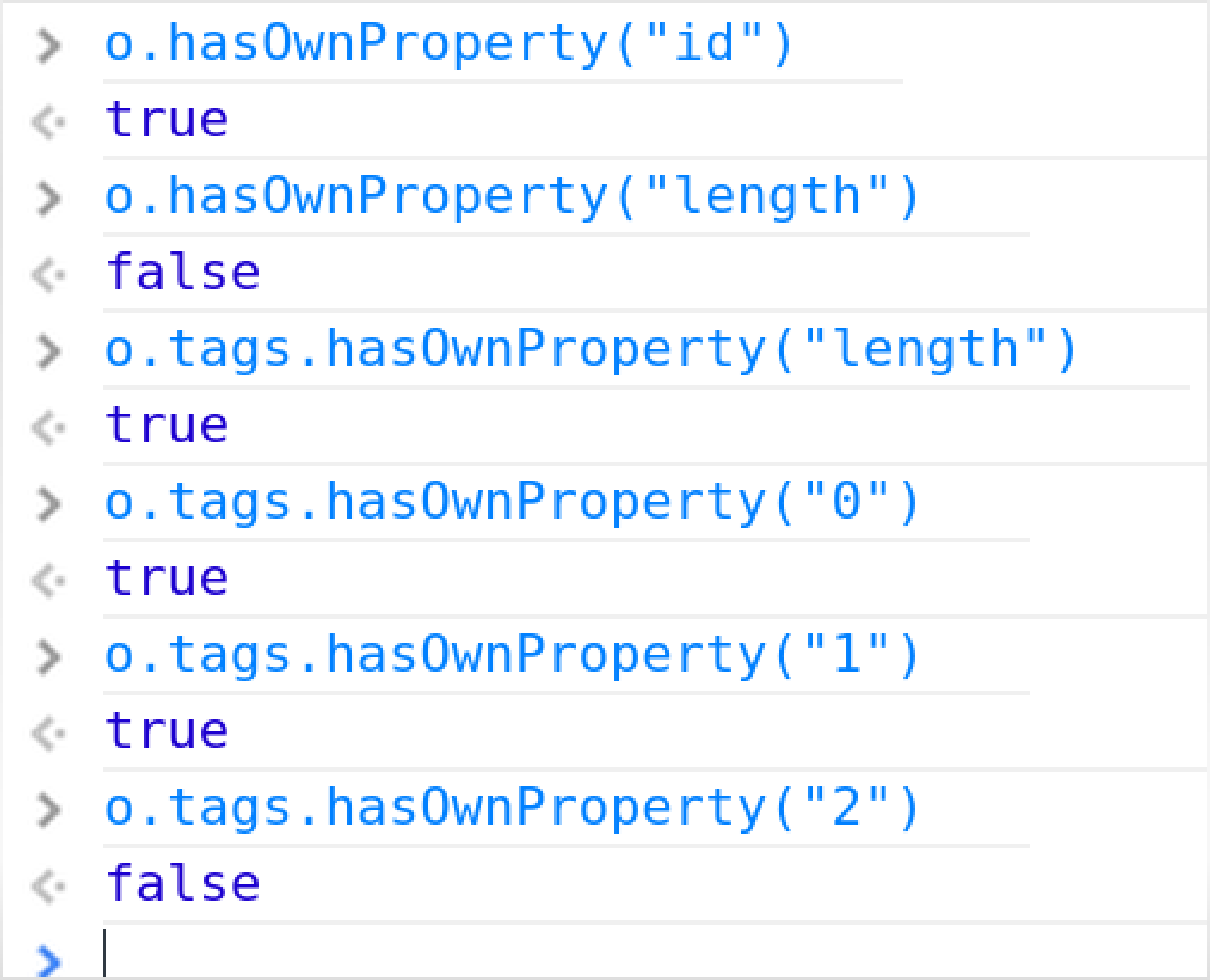


```
employee
▼ Object {job: "Js Developer", tags: Array[2], name: "Narendra", place: "Delhi"}
    job: "Js Developer"
  ▶ tags: Array[2]
  ▼ __proto__: Object
      name: "Narendra"
      place: "Delhi"
    ▼ __proto__: Object
      ▶ __defineGetter__: function __defineGetter__() { [native code] }
      ▶ __defineSetter__: function __defineSetter__() { [native code] }
      ▶ __lookupGetter__: function __lookupGetter__() { [native code] }
      ▶ __lookupSetter__: function __lookupSetter__() { [native code] }
      ▶ constructor: function Object() { [native code] }
      ▶ hasOwnProperty: function hasOwnProperty() { [native code] }
      ▶ isPrototypeOf: function isPrototypeOf() { [native code] }
      ▶ propertyIsEnumerable: function propertyIsEnumerable() { [native code] }
      ▶ toLocaleString: function toLocaleString() { [native code] }
      ▶ toString: function toString() { [native code] }
      ▶ valueOf: function valueOf() { [native code] }
      ▶ get __proto__: function __proto__() { [native code] }
      ▶ set __proto__: function __proto__() { [native code] }
```



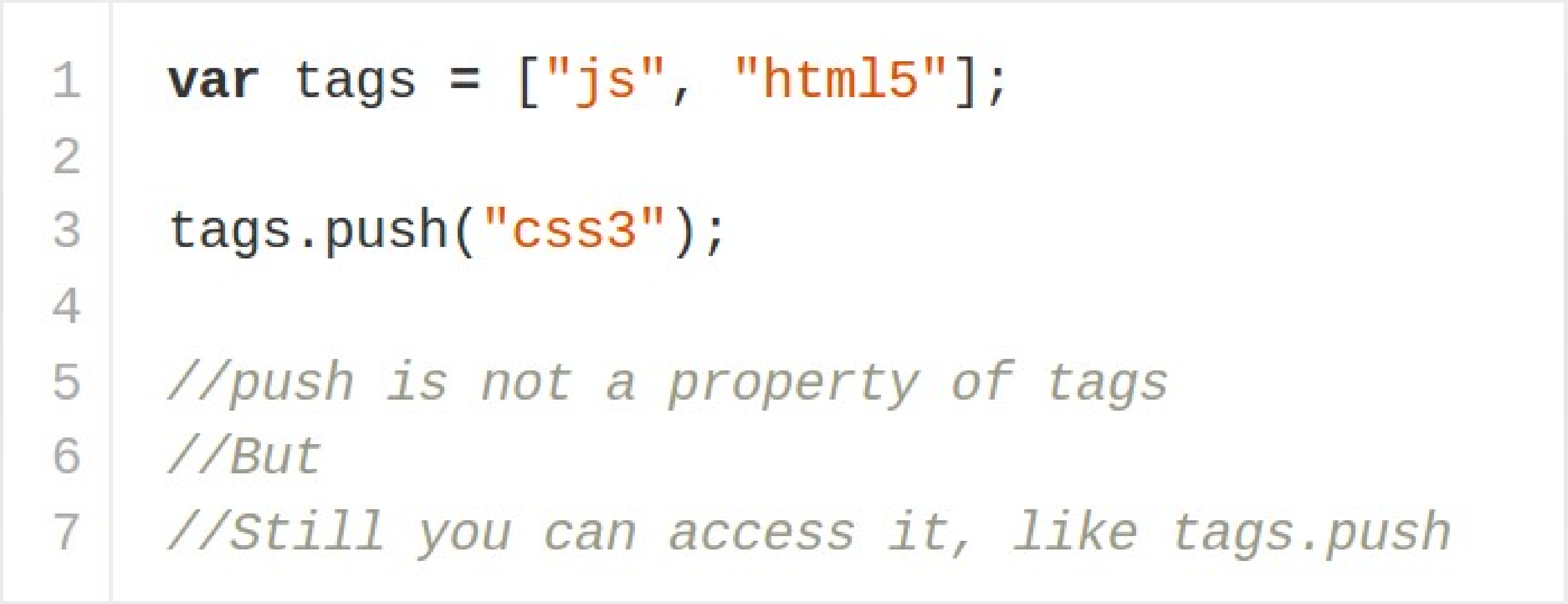
```
▼ tags: Array[2]
    0: "js"
    1: "html5"
    length: 2
  ▼   proto  : Array[0]
    ▶ concat: function concat() { [native code] }
    ▶ constructor: function Array() { [native code] }
    ▶ every: function every() { [native code] }
    ▶ filter: function filter() { [native code] }
    ▶ forEach: function forEach() { [native code] }
    ▶ indexOf: function indexOf() { [native code] }
    ▶ join: function join() { [native code] }
    ▶ lastIndexOf: function lastIndexOf() { [native code] }
      length: 0
    ▶ map: function map() { [native code] }
    ▶ pop: function pop() { [native code] }
    ▶ push: function push() { [native code] }
    ▶ reduce: function reduce() { [native code] }
    ▶ reduceRight: function reduceRight() { [native code] }
    ▶ reverse: function reverse() { [native code] }
    ▶ shift: function shift() { [native code] }
    ▶ slice: function slice() { [native code] }
    ▶ some: function some() { [native code] }
    ▶ sort: function sort() { [native code] }
    ▶ splice: function splice() { [native code] }
    ▶ toLocaleString: function toLocaleString() { [native code] }
    ▶ toString: function toString() { [native code] }
    ▶ unshift: function unshift() { [native code] }
    ▼   proto  : Object
      ▶   defineGetter  : function   defineGetter  () { [na
      ▶   defineSetter  : function   defineSetter  () { [native code] }
      ▶   lookupGetter  : function   lookupGetter  () { [native code] }
      ▶   lookupSetter  : function   lookupSetter  () { [native code] }
      ▶ constructor: function Object() { [native code] }
```

hasOwnProperty

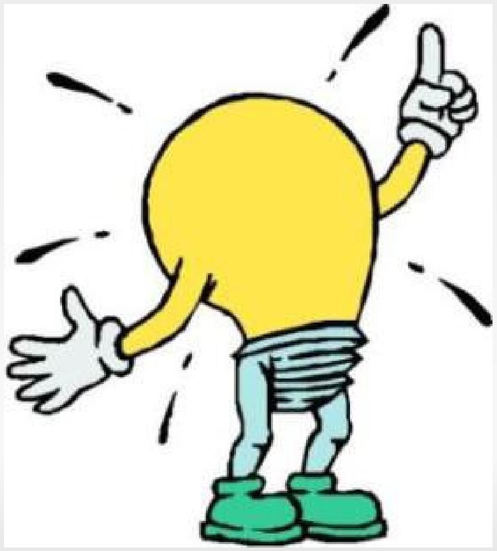


```
> o.hasOwnProperty("id")
<· true
> o.hasOwnProperty("length")
<· false
> o.tags.hasOwnProperty("length")
<· true
> o.tags.hasOwnProperty("0")
<· true
> o.tags.hasOwnProperty("1")
<· true
> o.tags.hasOwnProperty("2")
<· false
>
```

Now you know! Prototype Chaining

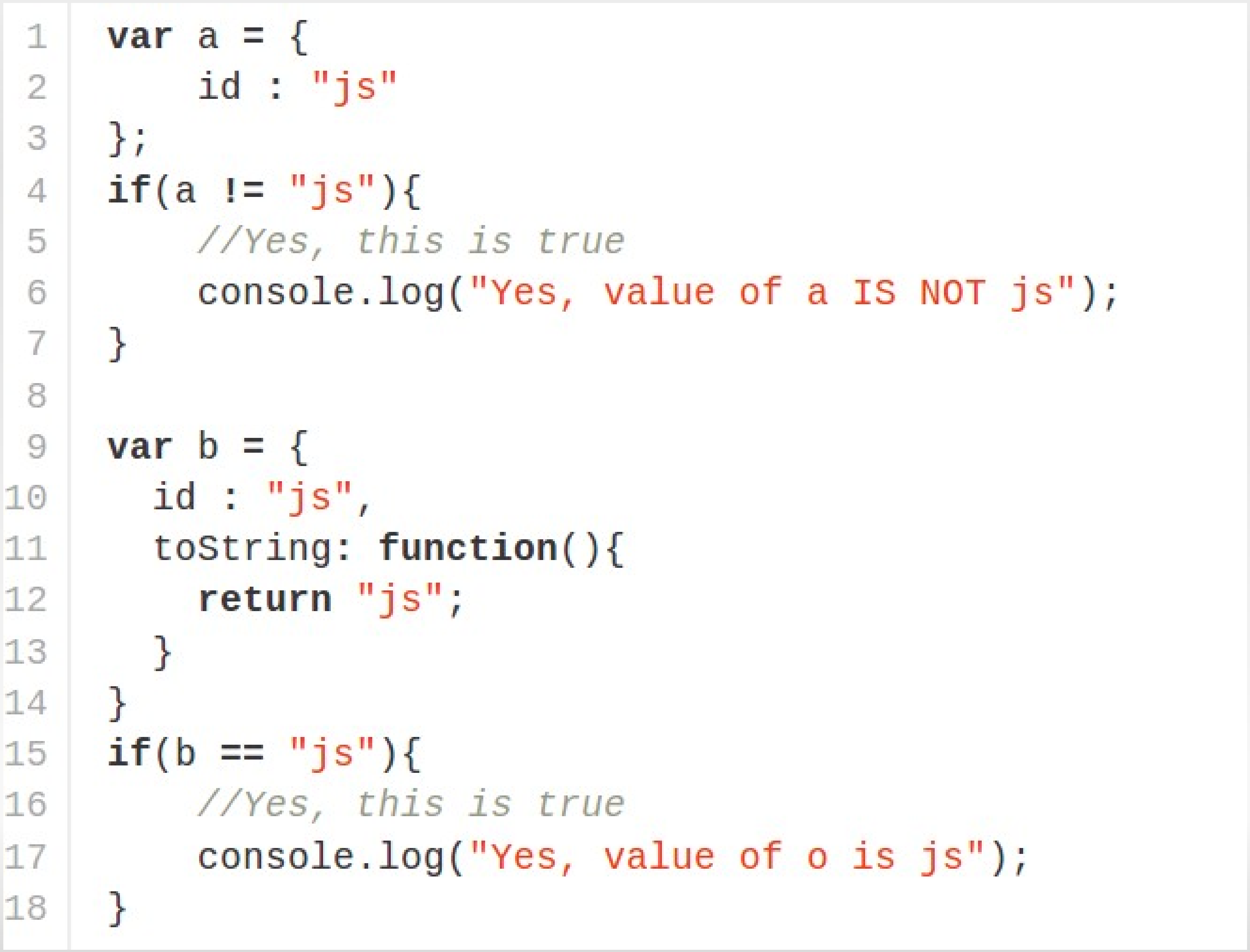


```
var tags = ["js", "html5"];

tags.push("css3");

//push is not a property of tags
//But
//Still you can access it, like tags.push
```



for a property, first it will look into ***object***,
if unable to find, it will look into
object.__proto__ ,
if unable to find, it will look into
object.__proto__.proto__ and so on,
until it find null. This is call **Prototype Chaining**

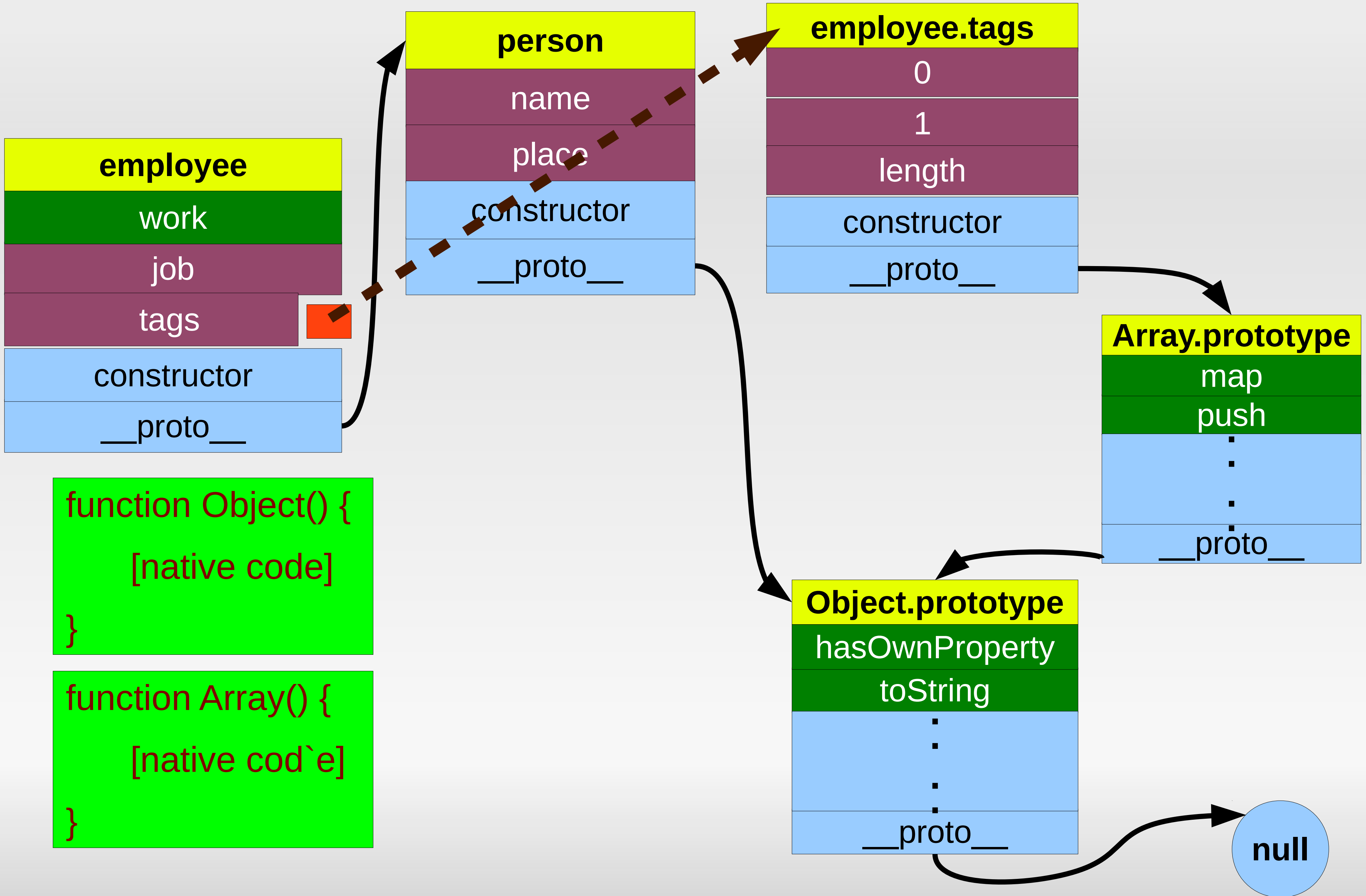
Over-riding properties !



```javascript
var a = {
    id : "js"
};
if(a != "js"){
    //Yes, this is true
    console.log("Yes, value of a IS NOT js");
}

var b = {
  id : "js",
  toString: function(){
    return "js";
  }
}
if(b == "js"){
    //Yes, this is true
    console.log("Yes, value of o is js");
}
```

Object Linking Diagrams

Longer the prototype chain, more the access time

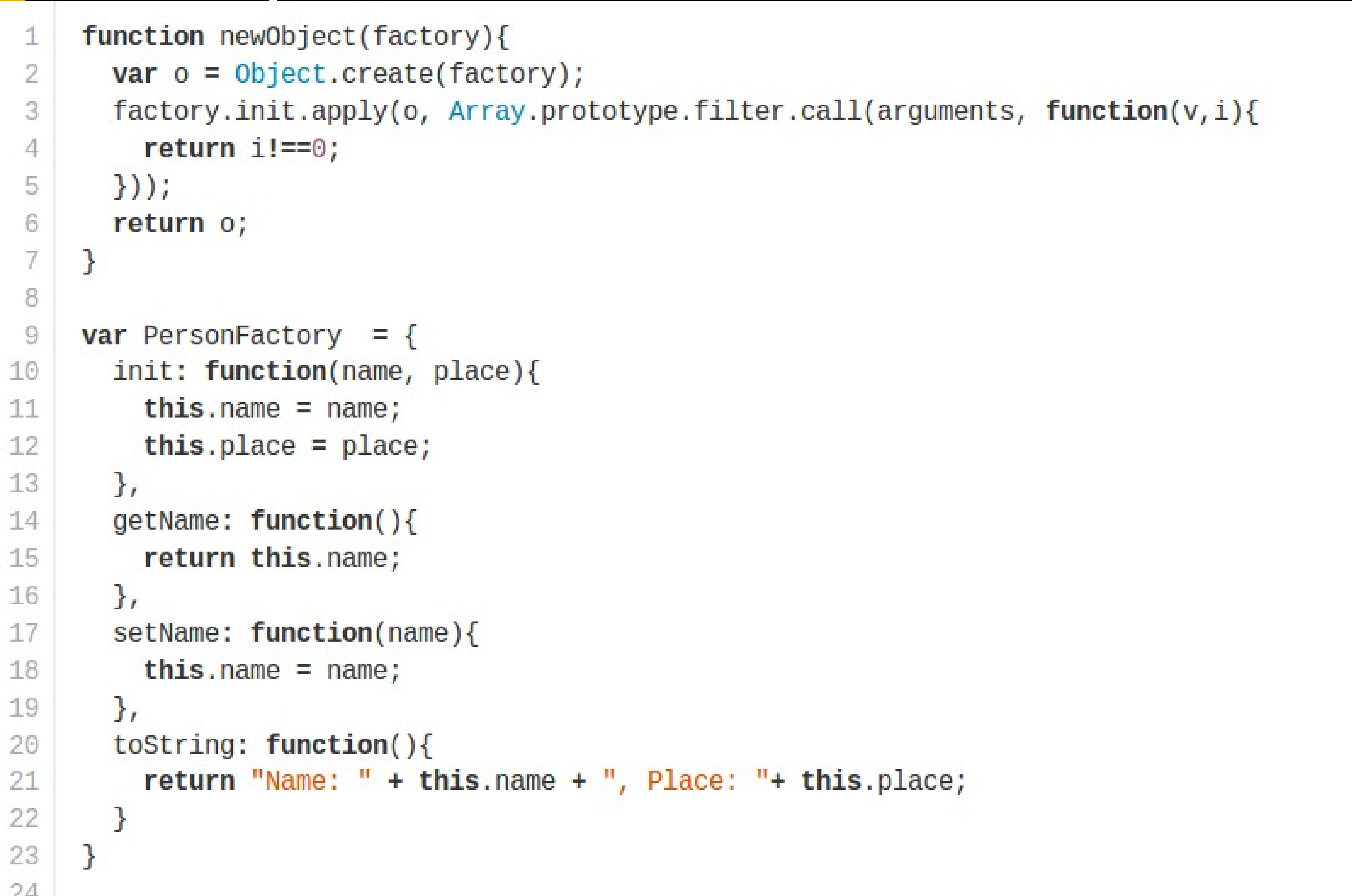


```
function Object() {
      [native code]
}
```

```
function Array() {
      [native cod`e]
}
```

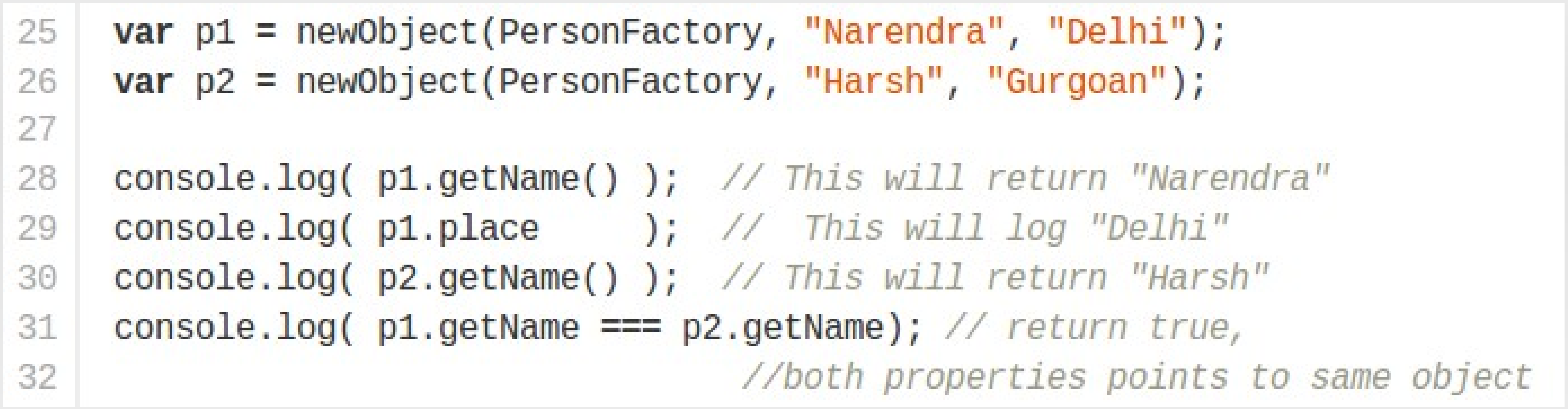
But I cannot see multiple object ?

Here you are



```javascript
function newObject(factory){
  var o = Object.create(factory);
  factory.init.apply(o, Array.prototype.filter.call(arguments, function(v,i){
    return i!==0;
  }));
  return o;
}

var PersonFactory  = {
  init: function(name, place){
    this.name = name;
    this.place = place;
  },
  getName: function(){
    return this.name;
  },
  setName: function(name){
    this.name = name;
  },
  toString: function(){
    return "Name: " + this.name + ", Place: "+ this.place;
  }
}
```

Here you are

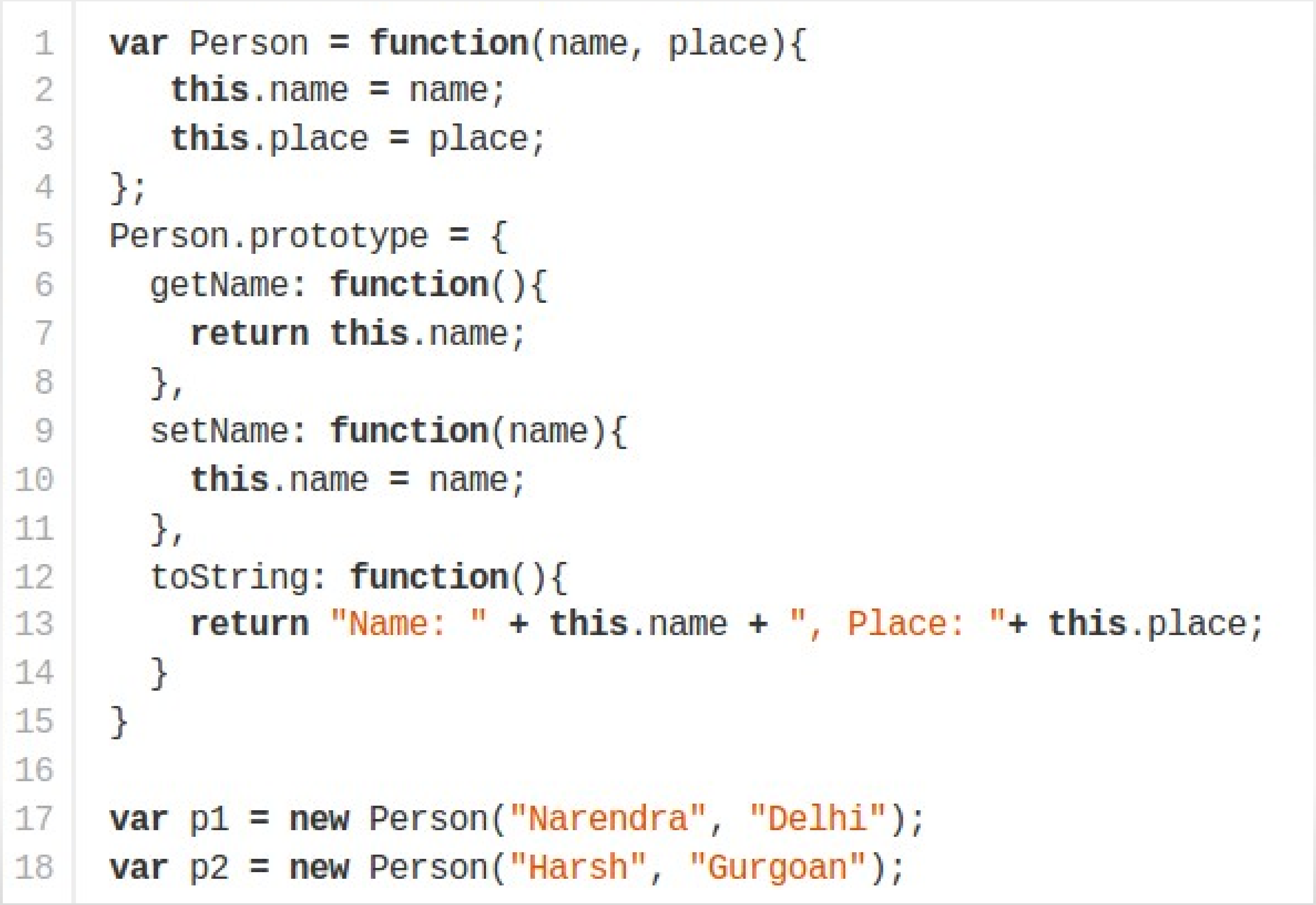


```
var p1 = newObject(PersonFactory, "Narendra", "Delhi");
var p2 = newObject(PersonFactory, "Harsh", "Gurgoan");

console.log( p1.getName() );  // This will return "Narendra"
console.log( p1.place     );  //  This will log "Delhi"
console.log( p2.getName() );  // This will return "Harsh"
console.log( p1.getName === p2.getName); // return true,
                               //both properties points to same object
```

Advantage Each Object point to same set of methods

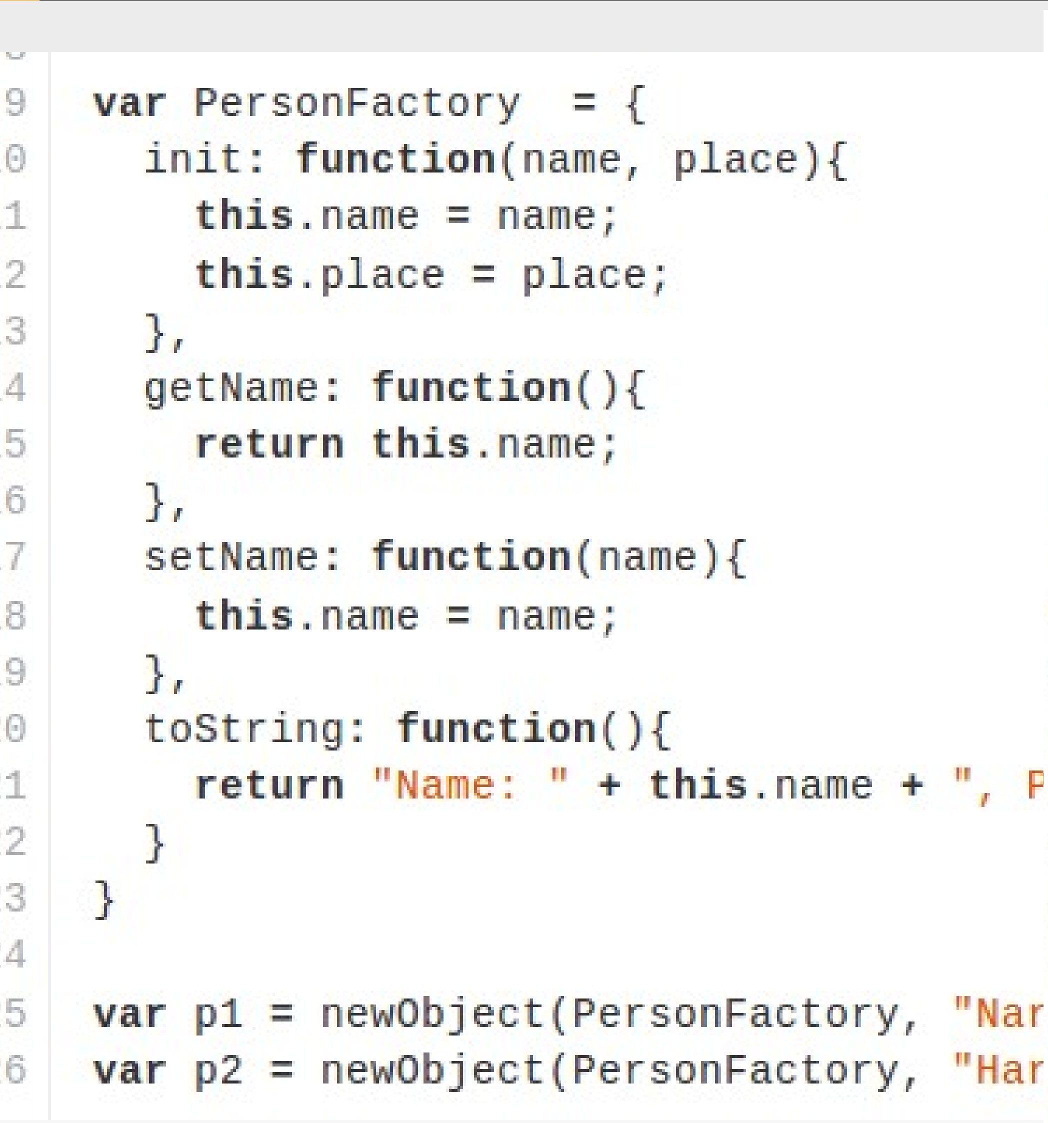
Disadvantage properties are public, easily accessible from outside
Ex - p1.place

new Keyword - Alternate Syntax

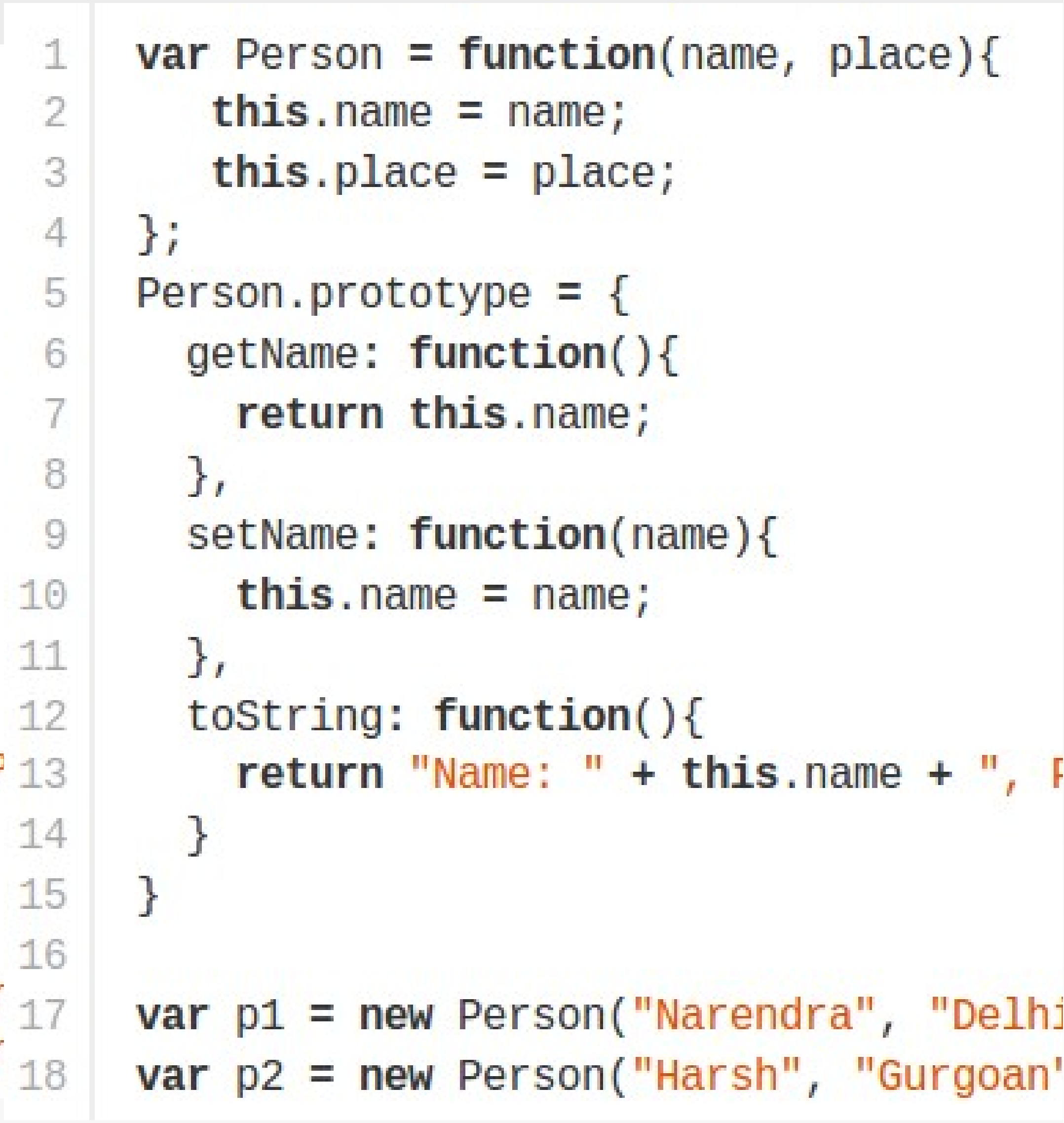


```
var Person = function(name, place){
   this.name = name;
   this.place = place;
};
Person.prototype = {
  getName: function(){
    return this.name;
  },
  setName: function(name){
    this.name = name;
  },
  toString: function(){
    return "Name: " + this.name + ", Place: "+ this.place;
  }
}

var p1 = new Person("Narendra", "Delhi");
var p2 = new Person("Harsh", "Gurgoan");
```

Let compare, both syntax



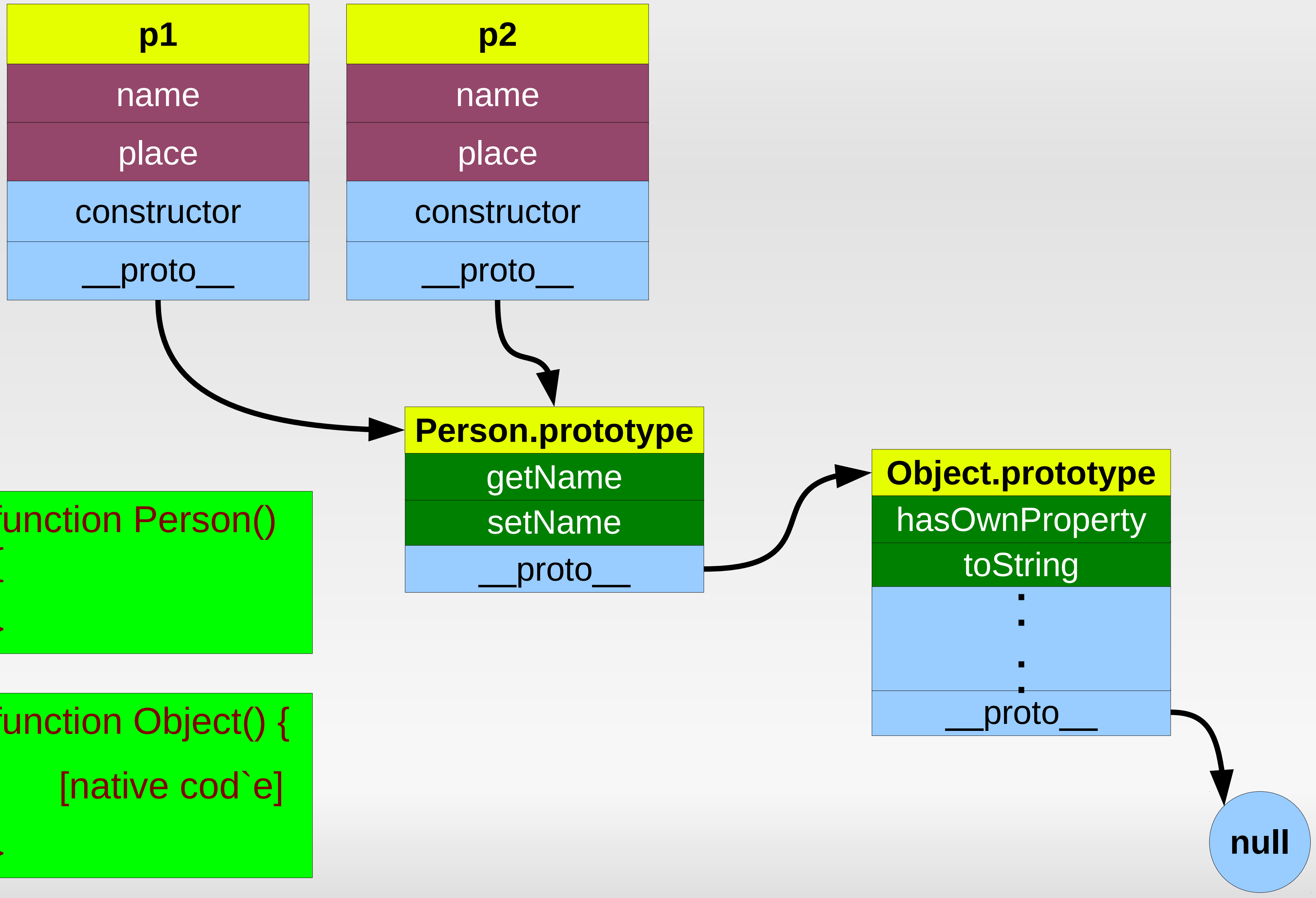
```javascript
var PersonFactory  = {
  init: function(name, place){
    this.name = name;
    this.place = place;
  },
  getName: function(){
    return this.name;
  },
  setName: function(name){
    this.name = name;
  },
  toString: function(){
    return "Name: " + this.name + ", P
  }
}

var p1 = newObject(PersonFactory, "Nar
var p2 = newObject(PersonFactory, "Har
```



```javascript
var Person = function(name, place){
   this.name = name;
   this.place = place;
};
Person.prototype = {
  getName: function(){
    return this.name;
  },
  setName: function(name){
    this.name = name;
  },
  toString: function(){
    return "Name: " + this.name + ", P
  }
}

var p1 = new Person("Narendra", "Delhi
var p2 = new Person("Harsh", "Gurgoan'
```

Object Linking Diagrams

Longer the prototype chain, more the access time

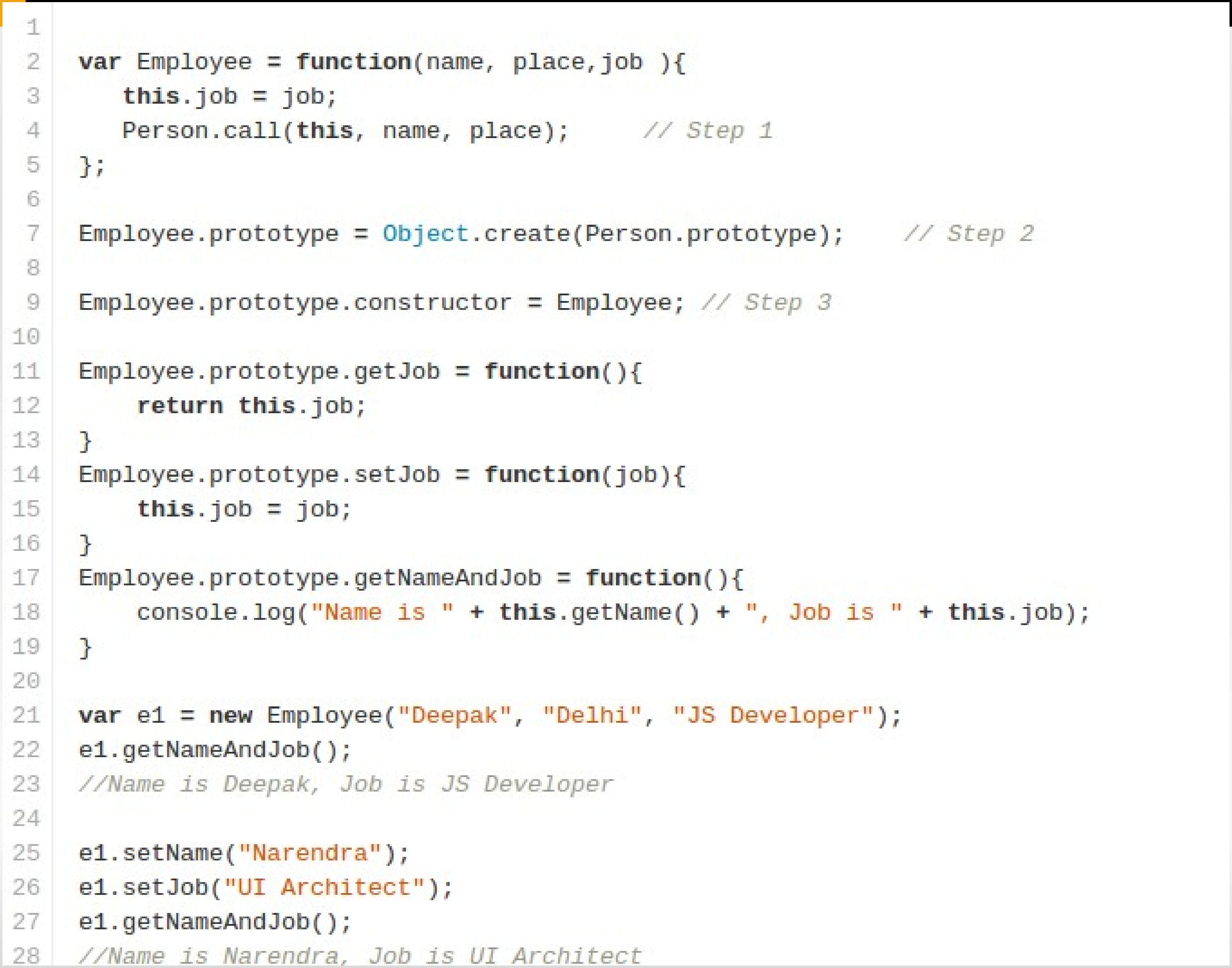


```
function Person()
{
}
```

```
function Object() {
      [native cod`e]
}
```

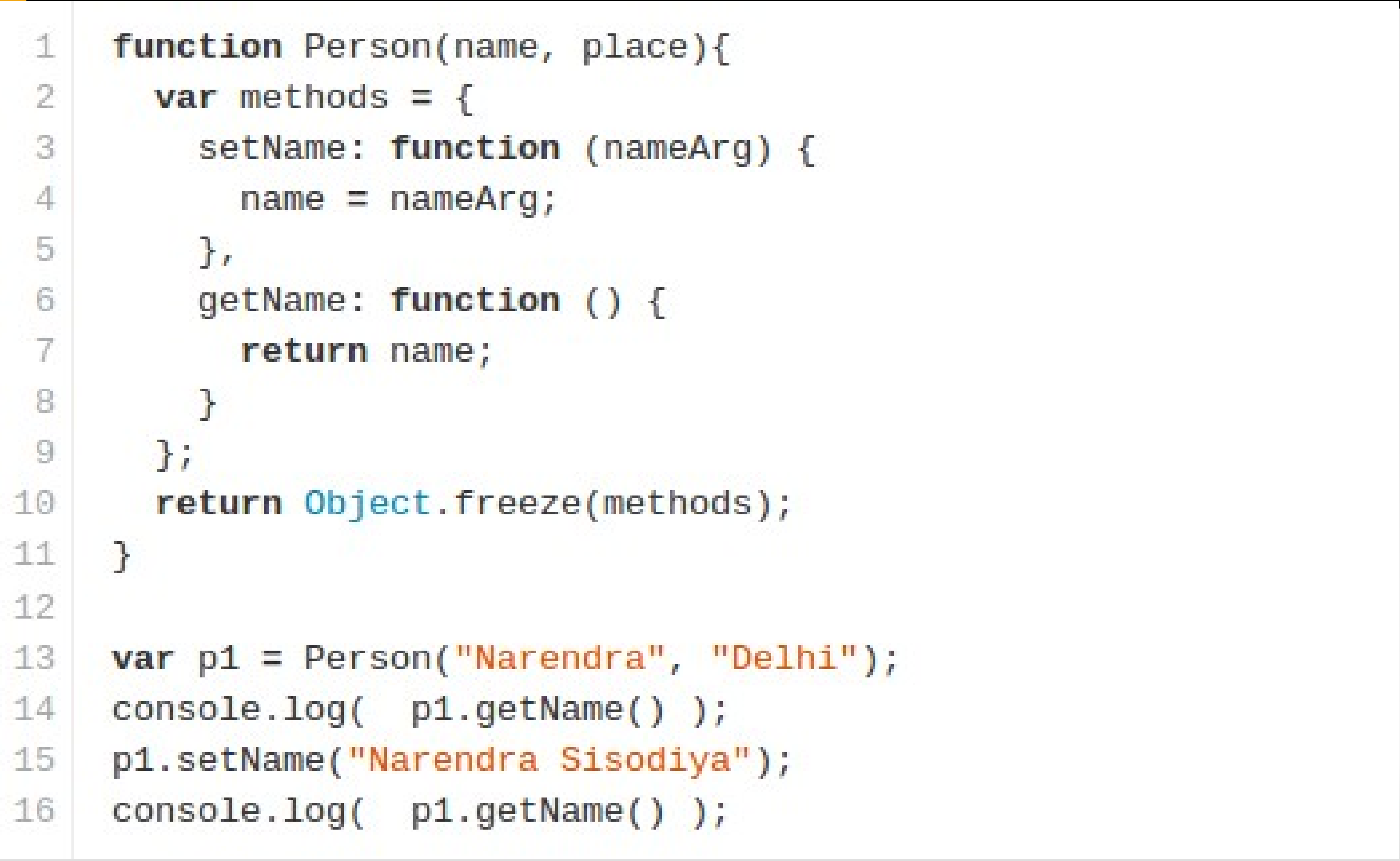
Understanding “this” keyword

Inheritance



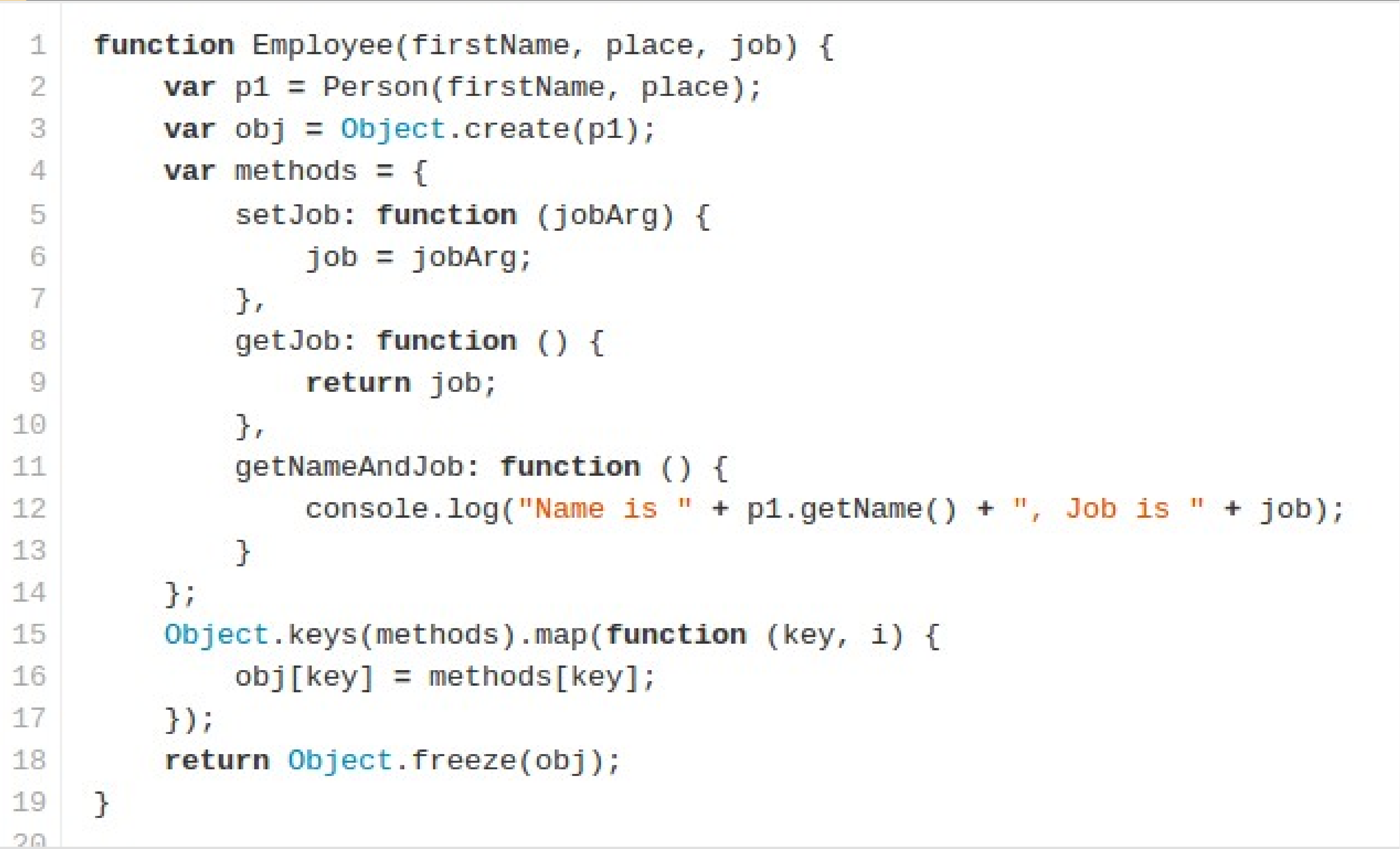
```

var Employee = function(name, place,job ){
   this.job = job;
   Person.call(this, name, place);     // Step 1
};

Employee.prototype = Object.create(Person.prototype);    // Step 2

Employee.prototype.constructor = Employee; // Step 3

Employee.prototype.getJob = function(){
    return this.job;
}
Employee.prototype.setJob = function(job){
    this.job = job;
}
Employee.prototype.getNameAndJob = function(){
    console.log("Name is " + this.getName() + ", Job is " + this.job);
}

var e1 = new Employee("Deepak", "Delhi", "JS Developer");
e1.getNameAndJob();
//Name is Deepak, Job is JS Developer

e1.setName("Narendra");
e1.setJob("UI Architect");
e1.getNameAndJob();
//Name is Narendra, Job is UI Architect
```

Without new and this keyword



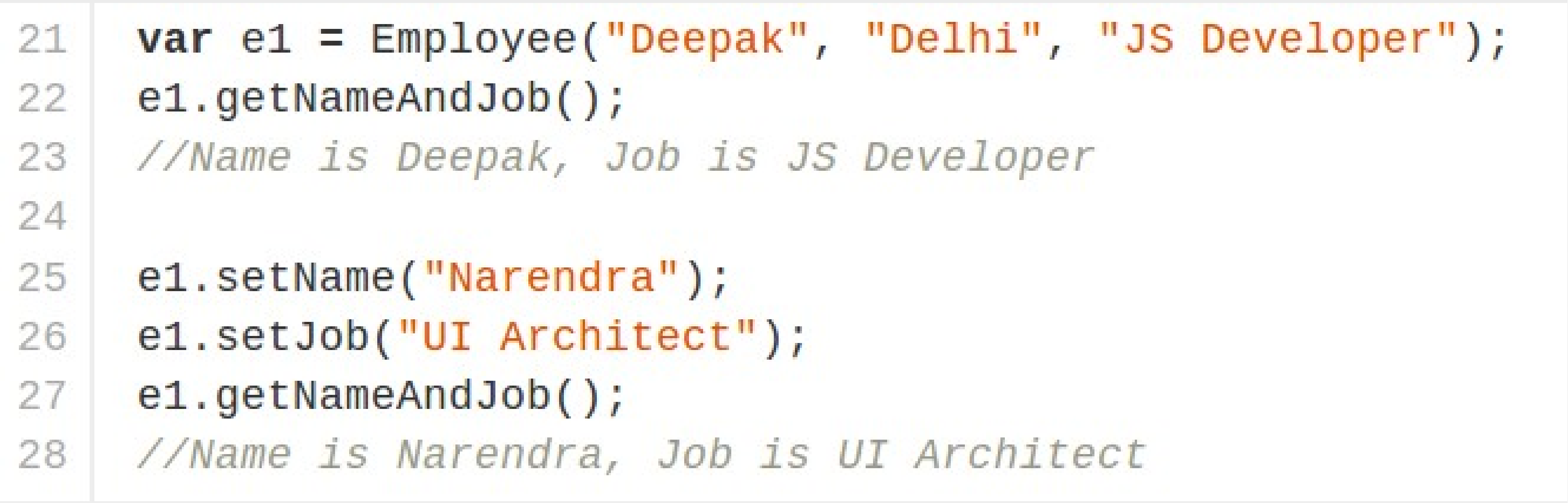
```
function Person(name, place){
  var methods = {
    setName: function (nameArg) {
      name = nameArg;
    },
    getName: function () {
      return name;
    }
  };
  return Object.freeze(methods);
}

var p1 = Person("Narendra", "Delhi");
console.log(  p1.getName() );
p1.setName("Narendra Sisodiya");
console.log(  p1.getName() );
```

Inheritance



```
function Employee(firstName, place, job) {
    var p1 = Person(firstName, place);
    var obj = Object.create(p1);
    var methods = {
        setJob: function (jobArg) {
            job = jobArg;
        },
        getJob: function () {
            return job;
        },
        getNameAndJob: function () {
            console.log("Name is " + p1.getName() + ", Job is " + job);
        }
    };
    Object.keys(methods).map(function (key, i) {
        obj[key] = methods[key];
    });
    return Object.freeze(obj);
}
```

Inheritance



```
var e1 = Employee("Deepak", "Delhi", "JS Developer");
e1.getNameAndJob();
//Name is Deepak, Job is JS Developer

e1.setName("Narendra");
e1.setJob("UI Architect");
e1.getNameAndJob();
//Name is Narendra, Job is UI Architect
```

Questions?

Function at prototype chain & context

```
eat: function (){
            alert(this.name +" is eating");
      }
```

function eat is not part of object **“narendra”, this.name**
When you run, narendra.eat(), eat() function of prototype chain will be executed with **Execution Context == narendra,**

Every function executed with a context, narendra.eat() will be executed with context as **“narendra”** so inside eat function, value of this will be narendra

```
      narendra === this       //true
```

```
person.eat();                        // Child of earth is eating
person.eat.call(narendra);       // Narendra Sisodiya is eating
```

What happen If I do not use “new” ?

```
var Car = function(data){
   console.dir(this);
   this.data = data;
}
var a = new Car();
var b = Car();
```

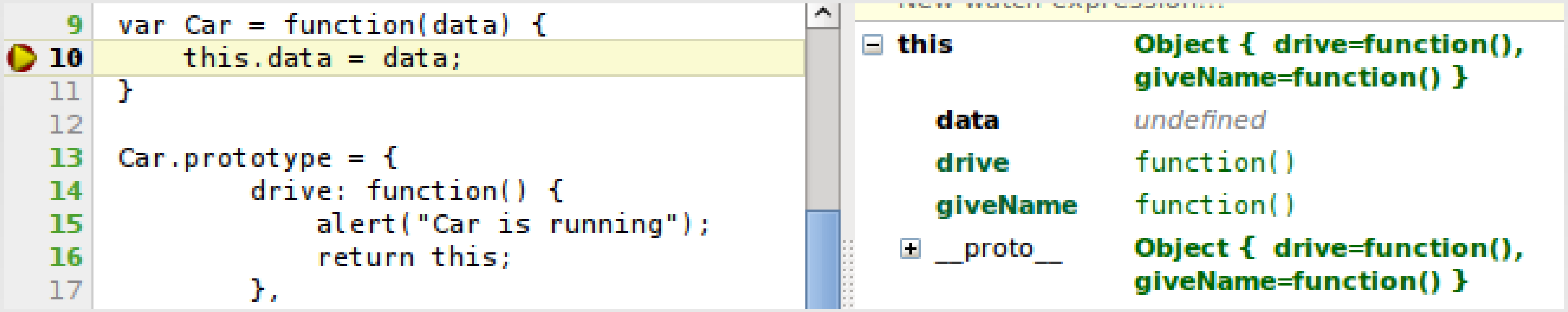
Without new – value of this will be
Window
object

WITH NEW

```
var Car = function(data){
   //var this = new Object.create(Car.prototype);
   console.dir(this);
   this.data = data;
   // return this
}
```

What happen If I do not use “new” ?

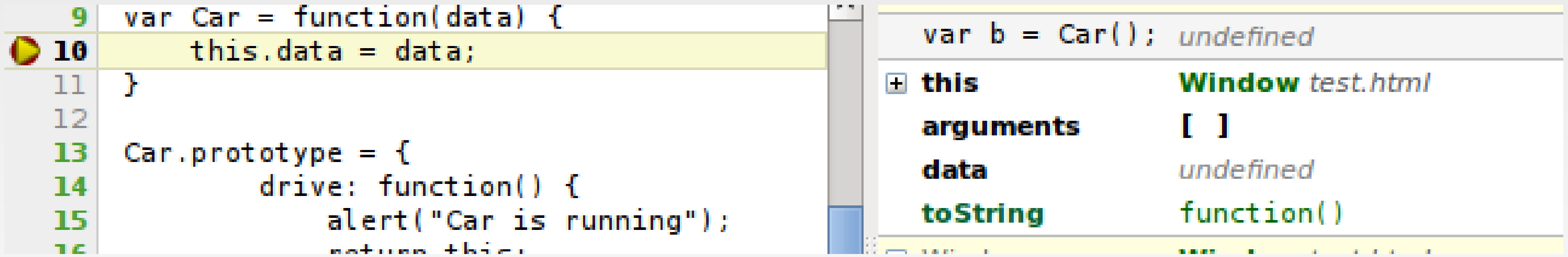
```
var a = new Car();
```



```
var Car = function(data) {
    this.data = data;
}

Car.prototype = {
        drive: function() {
            alert("Car is running");
            return this;
        },
```

```
this         Object { drive=function(), giveName=function() }
  data       undefined
  drive      function()
  giveName   function()
  __proto__  Object { drive=function(), giveName=function() }
```

```
var b = Car();
```



```
var Car = function(data) {
    this.data = data;
}

Car.prototype = {
        drive: function() {
            alert("Car is running");
```

```
var b = Car();   undefined
this             Window test.html
arguments        [ ]
data             undefined
toString         function()
```

When you invoke constructor with new operator, function will be passed with a THIS variable which inherit from function.prototype

Module Pattern without new Operator (1st way)

```
   var Car = function(data) {
      this.data = data;
   }

   Car.prototype = {
         drive: function() {
            alert("Car is running");
            return this;
         },
         giveName: function(){
            alert("The car name is " + this.data);
            return this;
         }
   };
var CarFactory = function(data){
   return new Car(data);
}
```