# Prototype

# and

# Inheritance

# Outline

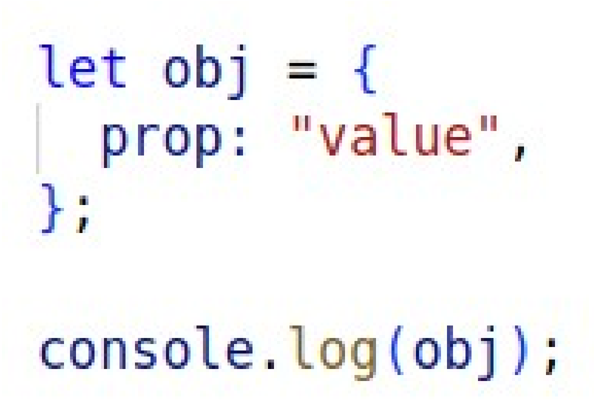
- Introduction
- Inheritance with the prototype chain
- Constructors
- Building large inheritance chain
- Examples
- Different ways to creating and mutating prototype chain

## Introduction

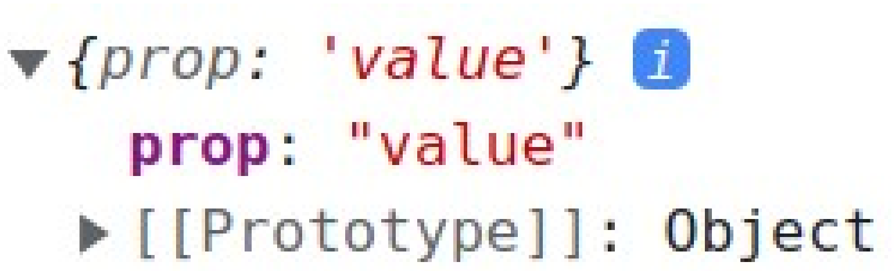
## Introduction

- When it comes to inheritance, JavaScript only has one construct: **objects**.
- Each object has a private property which holds a link to another object called its **prototype**.
- That prototype object has a prototype of its own, and so on until an object is reached with **null** as its prototype.
- By definition, **null** has no prototype, and acts as the final link in this prototype chain.
- It is possible to mutate any member of the prototype chain or even swap out the prototype at runtime, so concepts like static dispatching do not exist in JavaScript.

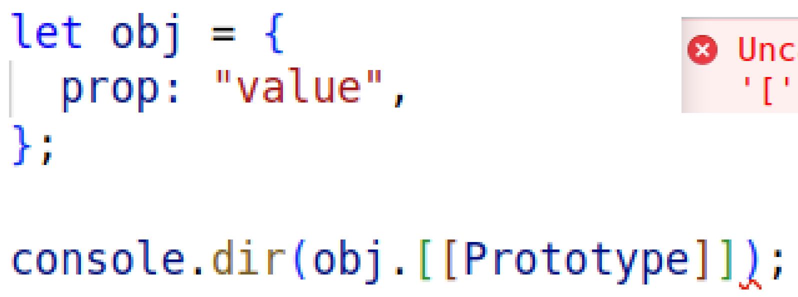
## Introduction



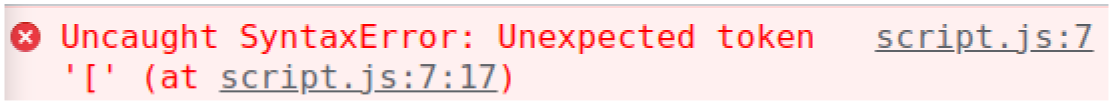
```
let obj = {
  prop: "value",
};

console.log(obj);
```



```
▼ {prop: 'value'} i
    prop: "value"
  ▶ [[Prototype]]: Object
```

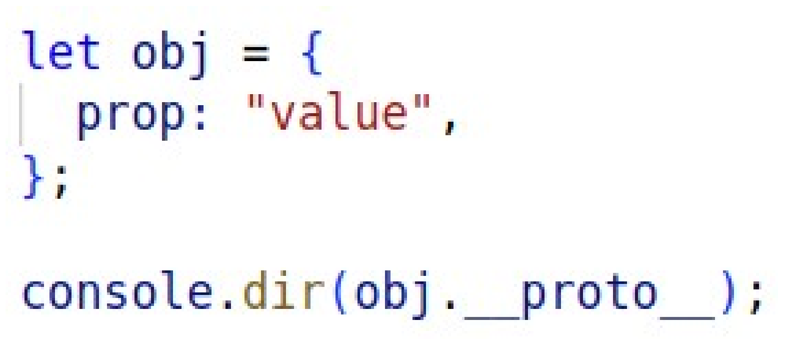


```
let obj = {
  prop: "value",
};

console.dir(obj.[[Prototype]]);
```

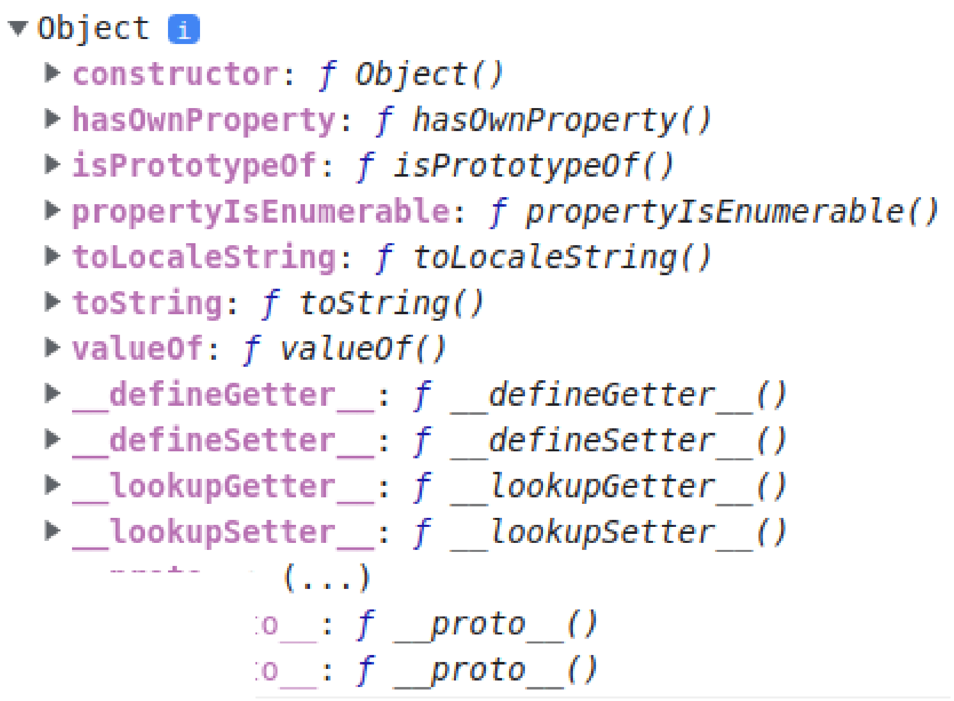


```
Uncaught SyntaxError: Unexpected token    script.js:7
'[' (at script.js:7:17)
```

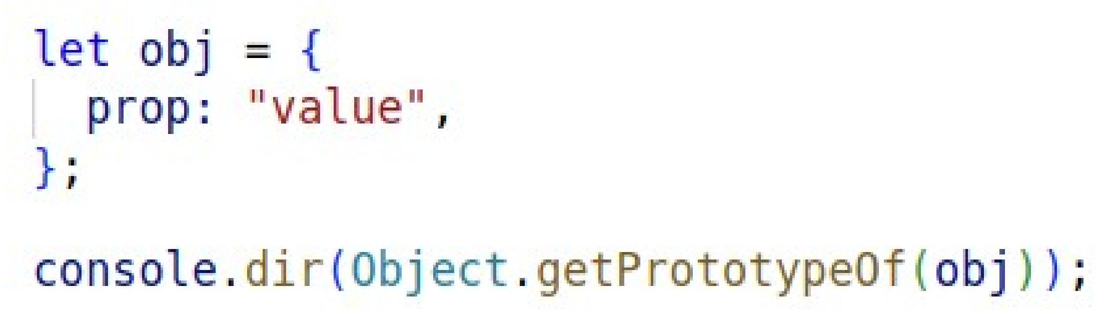
# Introduction



```
let obj = {
  prop: "value",
};

console.dir(obj.__proto__);
```



```
▼Object
  ▶constructor: ƒ Object()
  ▶hasOwnProperty: ƒ hasOwnProperty()
  ▶isPrototypeOf: ƒ isPrototypeOf()
  ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
  ▶toLocaleString: ƒ toLocaleString()
  ▶toString: ƒ toString()
  ▶valueOf: ƒ valueOf()
  ▶__defineGetter__: ƒ __defineGetter__()
  ▶__defineSetter__: ƒ __defineSetter__()
  ▶__lookupGetter__: ƒ __lookupGetter__()
  ▶__lookupSetter__: ƒ __lookupSetter__()
                     (...)
               o__: ƒ __proto__()
               o__: ƒ __proto__()
```



```
let obj = {
  prop: "value",
};

console.dir(Object.getPrototypeOf(obj));
```

## Accessing **[[Prototype]]** property

- Following the ECMAScript standard, the notation **someObject.[[Prototype]]** is used to designate the prototype of someObject.

- Since ECMAScript 2015, the **[[Prototype]]** is accessed using the accessors **Object.getPrototypeOf()** and **Object.setPrototypeOf()**.

- This is equivalent to the JavaScript accessor **__proto__** which is **non-standard** but de-facto implemented by many JavaScript engines.

- To prevent confusion while keeping it succinct, in our notation we will avoid using obj.__proto__ but use obj.[[Prototype]] instead. This corresponds to Object.getPrototypeOf(obj).

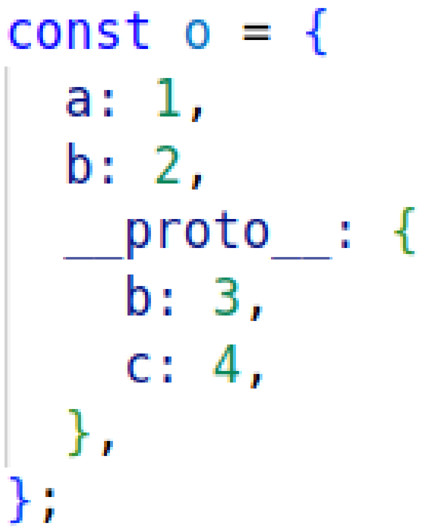
- It’s worth noting that the **{ __proto__: ... }** syntax is different from the **obj.__proto__** accessor: the former is standard and not deprecated.

## Inheritance with the prototype chain

## Inheriting Properties

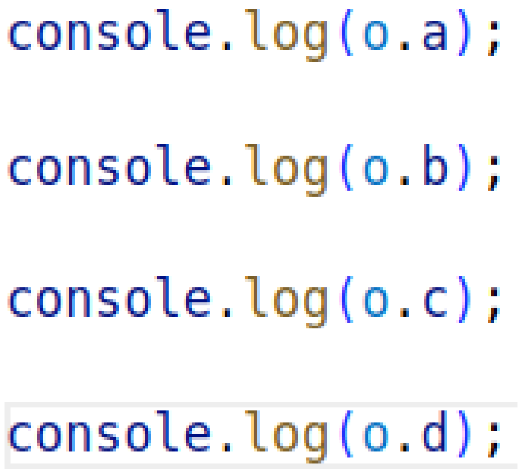
- JavaScript objects are dynamic "**bags**" of properties (referred to as own properties).
- JavaScript objects have a link to a prototype object.
- When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.

## Inheriting Properties

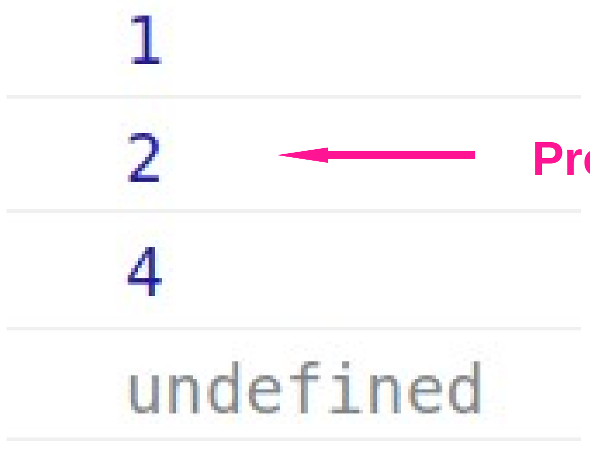


```
const o = {
  a: 1,
  b: 2,
  __proto__: {
    b: 3,
    c: 4,
  },
};
```

```
The full prototype chain looks like:

{ a: 1, b: 2 } --->

{ b: 3, c: 4 } --->

Object.prototype --->

null
```



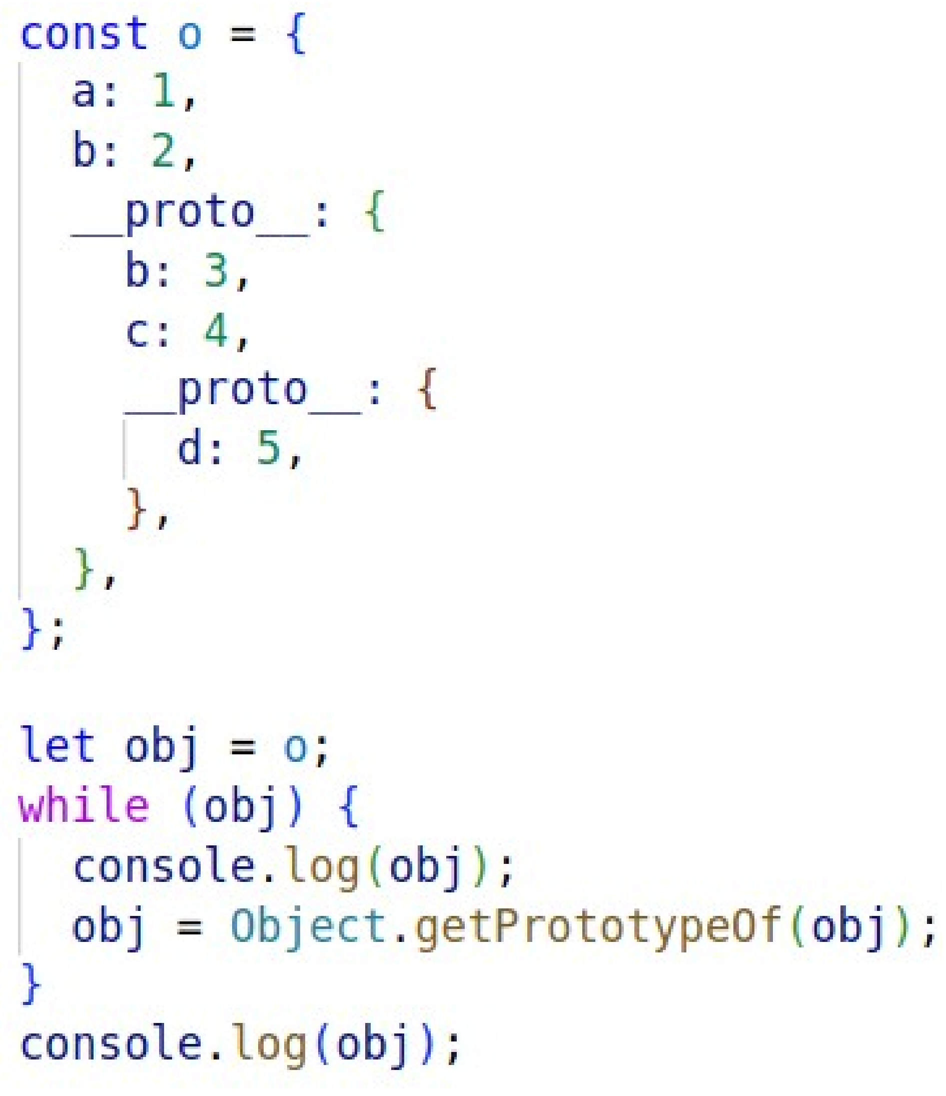
```
console.log(o.a);

console.log(o.b);

console.log(o.c);

console.log(o.d);
```



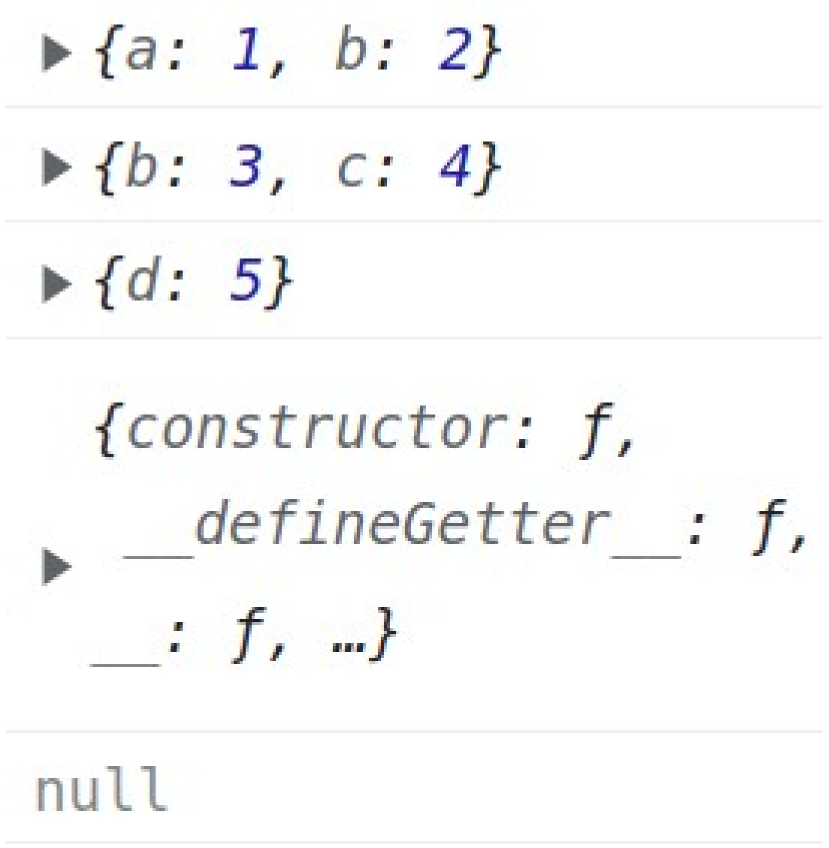
```
1
2
4
undefined
```

← **Property Shadowing**

# Inheriting Properties



```
const o = {
  a: 1,
  b: 2,
  __proto__: {
    b: 3,
    c: 4,
    __proto__: {
      d: 5,
    },
  },
};

let obj = o;
while (obj) {
  console.log(obj);
  obj = Object.getPrototypeOf(obj);
}
console.log(obj);
```

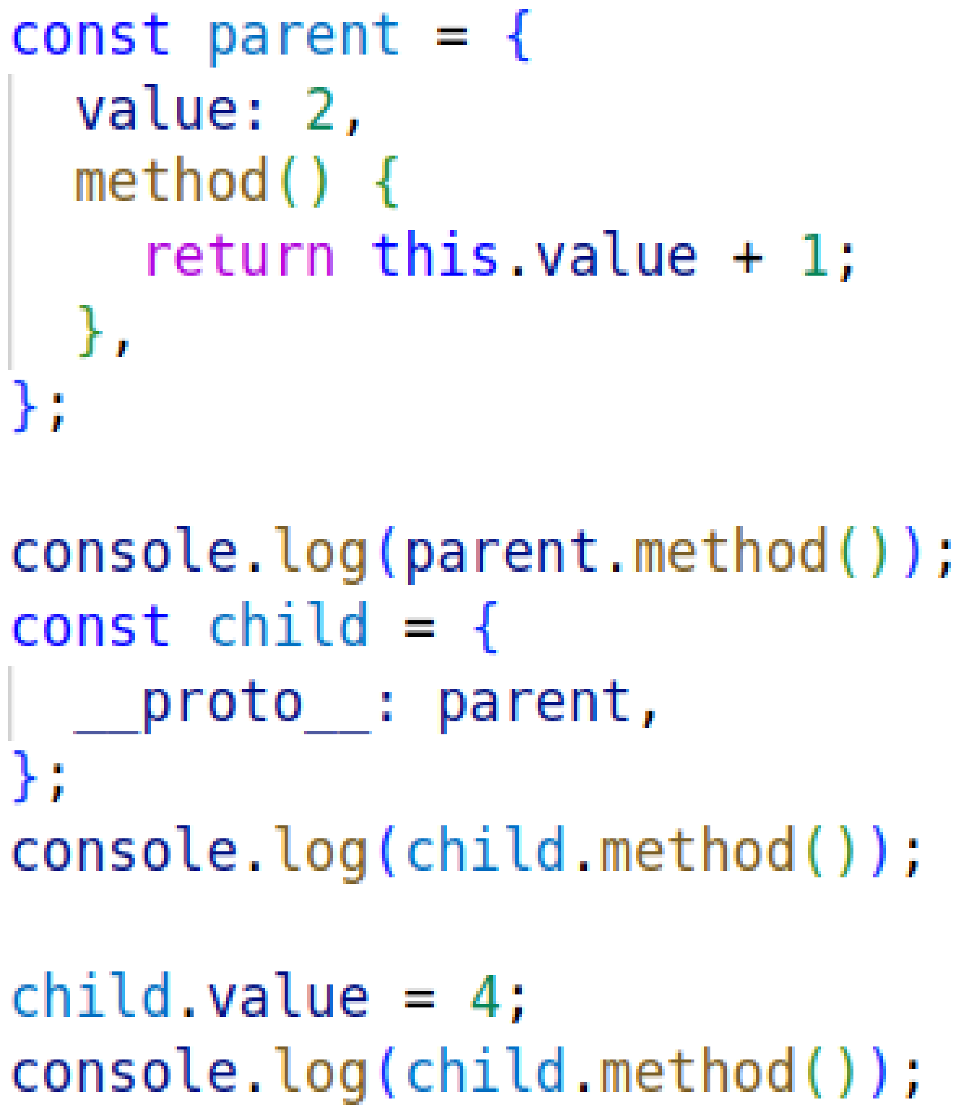


```
▶ {a: 1, b: 2}
▶ {b: 3, c: 4}
▶ {d: 5}
  {constructor: f,
▶  __defineGetter__: f,
 __: f, …}
null
```

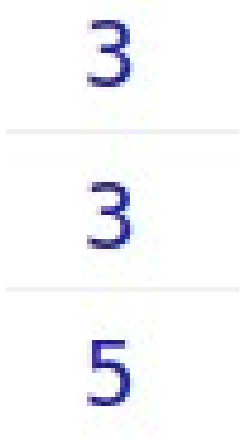
## Inheriting Methods

- JavaScript does not have "methods" in the form that class-based languages define them.
- In JavaScript, any function can be added to an object in the form of a property.
- An inherited function acts just as any other property, including property shadowing as shown above (in this case, a form of method overriding).
- When an inherited function is executed, the value of ***this*** points to the **inheriting** object, **not to the prototype** object where the function is an own property.

# Inheriting Methods



```
const parent = {
  value: 2,
  method() {
    return this.value + 1;
  },
};

console.log(parent.method());
const child = {
  __proto__: parent,
};
console.log(child.method());

child.value = 4;
console.log(child.method());
```

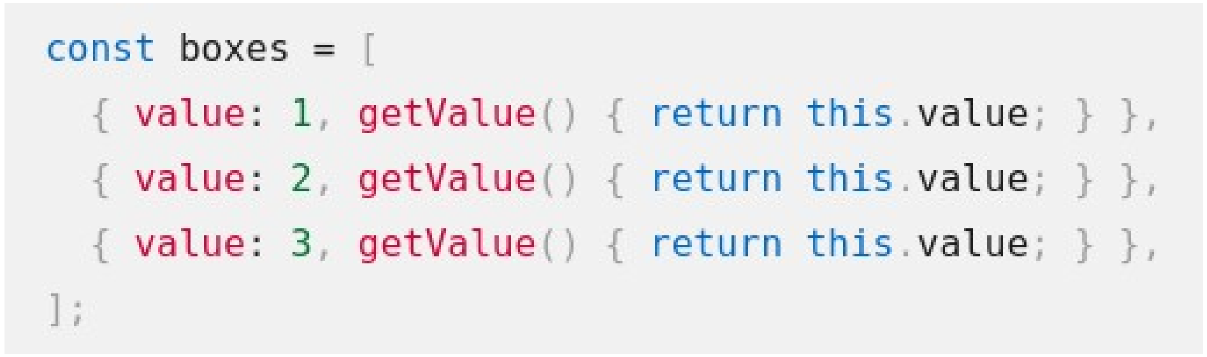


```
3
3
5
```

## Constructors

## Constructors

- The power of prototypes is that we can reuse a set of properties if they should be present on every instance — especially for methods.
- Suppose we are to create a series of boxes, where each box is an object that contains a value which can be accessed through a getValue function.
- A naive implementation would be:

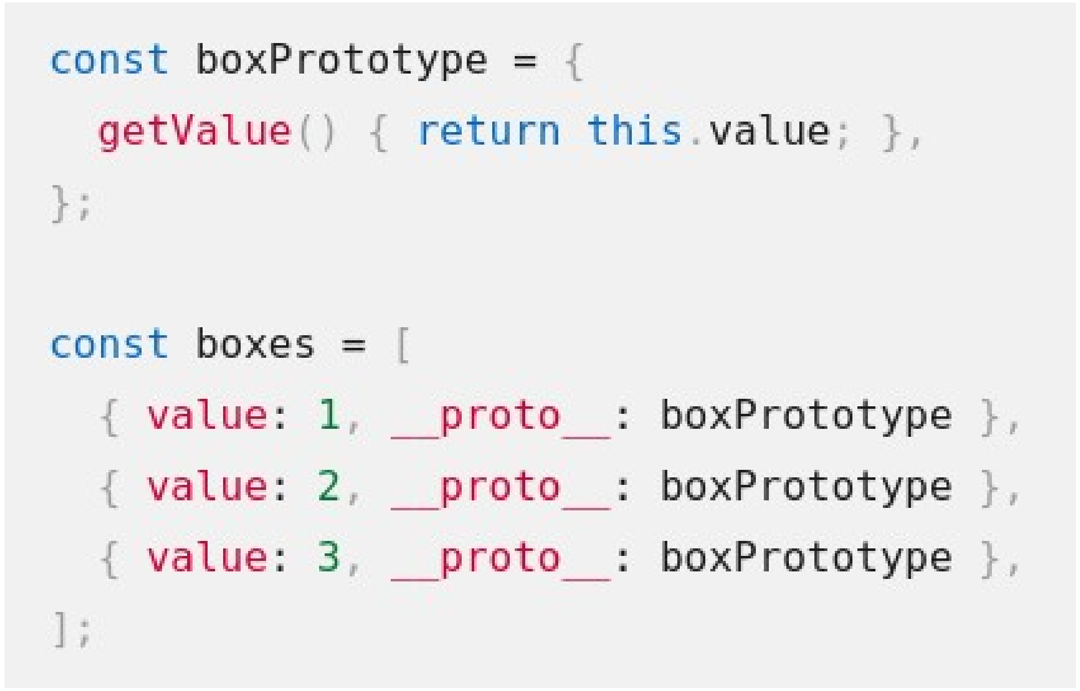


```
const boxes = [
  { value: 1, getValue() { return this.value; } },
  { value: 2, getValue() { return this.value; } },
  { value: 3, getValue() { return this.value; } },
];
```

- This is subpar, because each instance has its own function property that does the same thing, which is redundant and unnecessary.

## Constructors

- Instead, we can move getValue to the [[Prototype]] of all boxes:

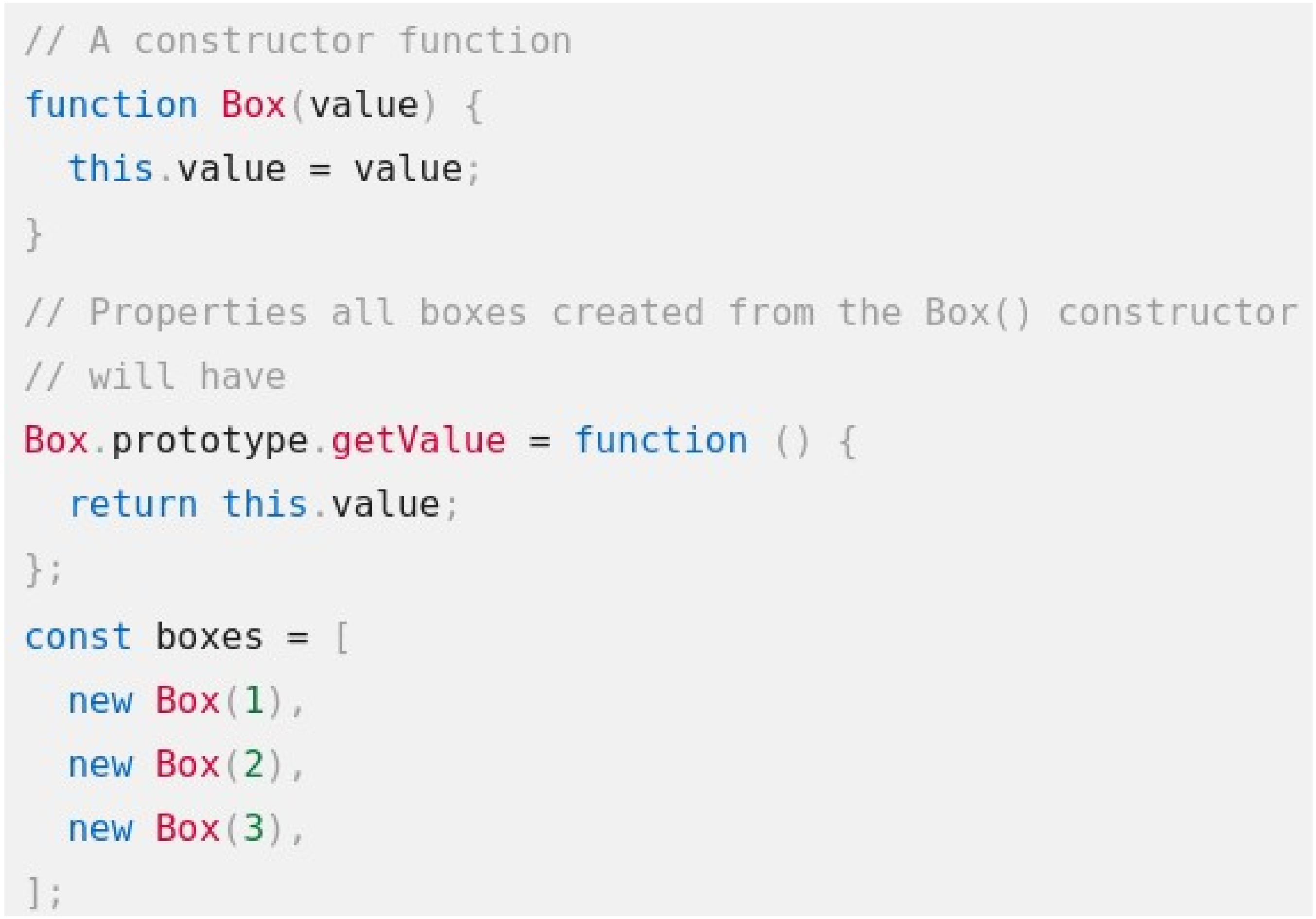


```
const boxPrototype = {
  getValue() { return this.value; },
};

const boxes = [
  { value: 1, __proto__: boxPrototype },
  { value: 2, __proto__: boxPrototype },
  { value: 3, __proto__: boxPrototype },
];
```

- This way, all boxes' getValue method will refer to the same function, lowering memory usage.
- However, manually binding the **__proto__** for every object creation is still very inconvenient.

# Constructors

- This is when we would use a constructor function, which automatically sets the [[Prototype]] for every object manufactured. Constructors are functions called with new.



```
// A constructor function
function Box(value) {
  this.value = value;
}
// Properties all boxes created from the Box() constructor
// will have
Box.prototype.getValue = function () {
  return this.value;
};
const boxes = [
  new Box(1),
  new Box(2),
  new Box(3),
];
```

# Constructors

- **Box.prototype** is not much different from the boxPrototype object we created previously — it's just a plain object.

- Every instance created from a constructor function will automatically have the constructor's **prototype** property as its **[[Prototype]]** — that is,

  **Object.getPrototypeOf(new Box()) === Box.prototype.**

- **Constructor.prototype** by default has one own property: **constructor**, which references the constructor function itself — that is,

  **Box.prototype.constructor === Box**

- This allows one to access the original constructor from any instance.

# Constructors

- **Constructor.prototype** is only useful when constructing instances.
- It has nothing to do with **Constructor.[[Prototype]]**, which is the constructor function's own prototype.
- Which is **Function.prototype.**
- That is,

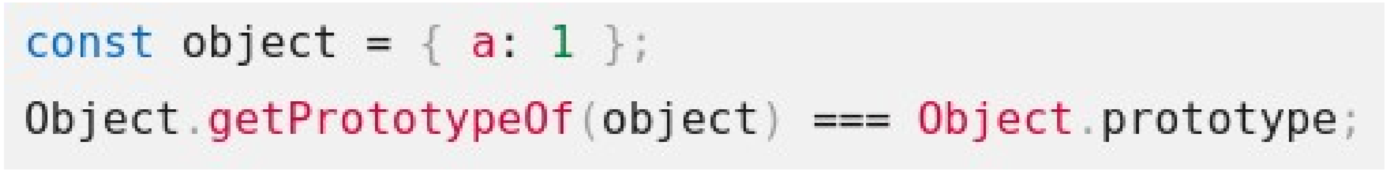
**Object.getPrototypeOf(Constructor)**

**===**

**Function.prototype**.

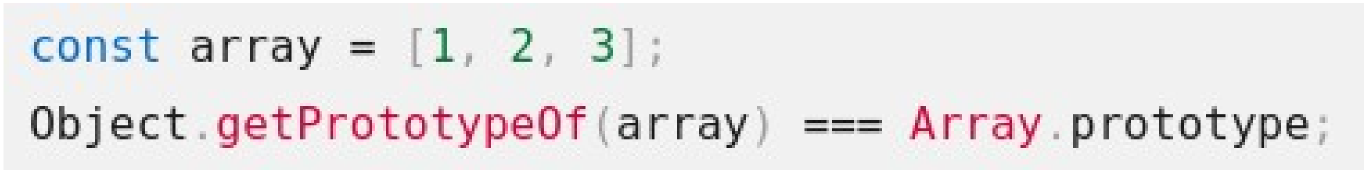
## Implicit constructors of literals

- Some literal syntaxes in JavaScript create instances that implicitly set the **[[Prototype]]**. For example:



```
const object = { a: 1 };
Object.getPrototypeOf(object) === Object.prototype;
```

**// true**

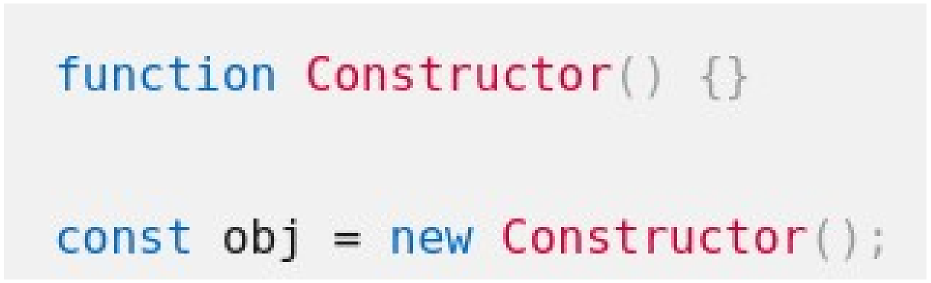


```
const array = [1, 2, 3];
Object.getPrototypeOf(array) === Array.prototype;
```

**// true**

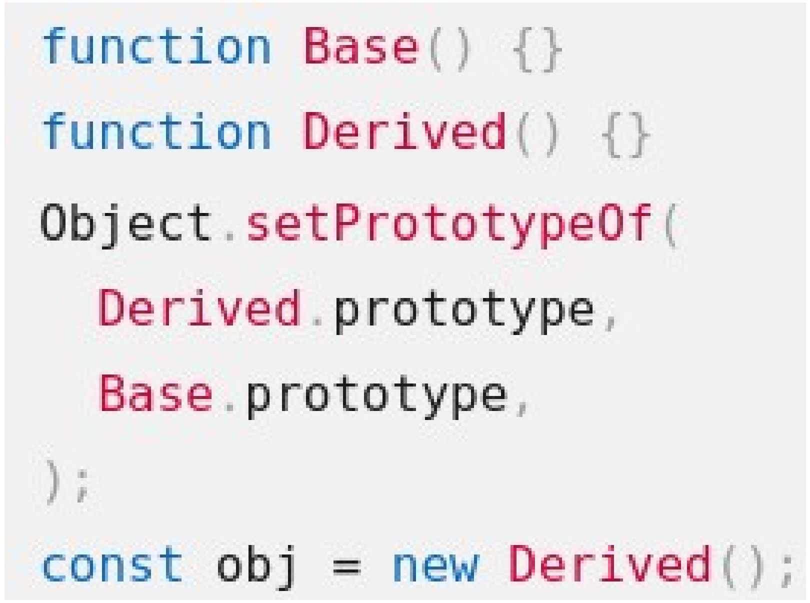
## Building longer inheritance chains

# Building longer inheritance chains



```
function Constructor() {}

const obj = new Constructor();
```

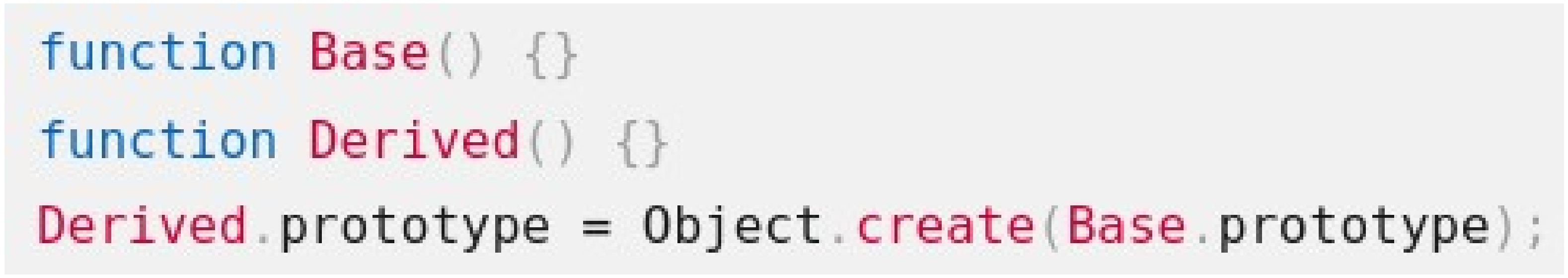
// obj ---> Constructor.prototype ---> Object.prototype ---> null



```
function Base() {}
function Derived() {}
Object.setPrototypeOf(
  Derived.prototype,
  Base.prototype,
);
const obj = new Derived();
```

// obj ---> Derived.prototype ---> Base.prototype ---> Object.prototype ---> null

# Building longer inheritance chains



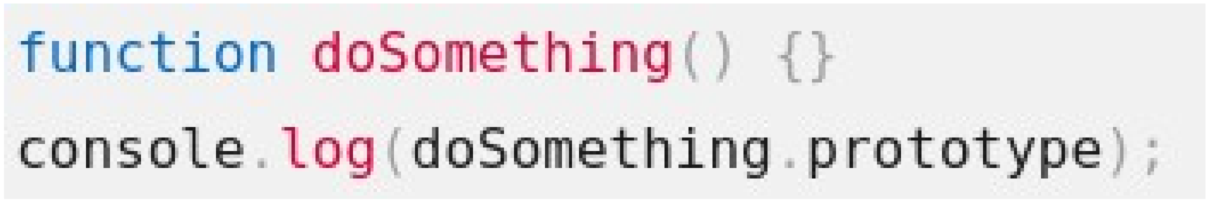
```
function Base() {}
function Derived() {}
Derived.prototype = Object.create(Base.prototype);
```

// obj ---> Derived.prototype ---> Base.prototype ---> Object.prototype ---> null

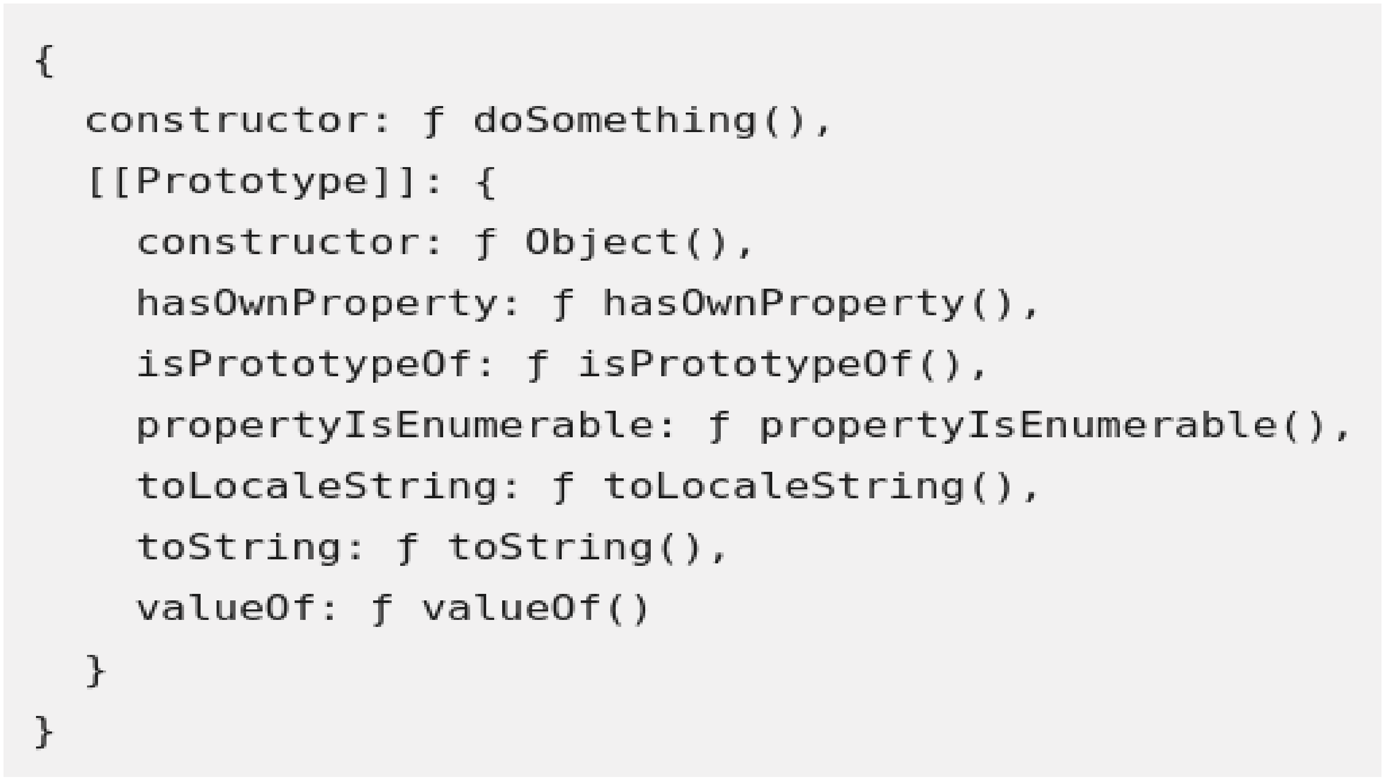
**However, because this re-assigns the prototype property, it's a bad practice.**

## Examples

## Example - 1

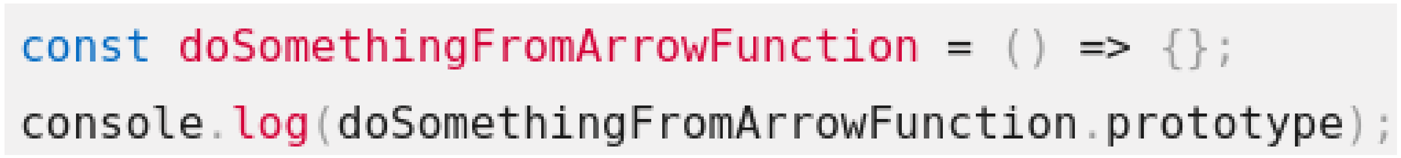


```
function doSomething() {}
console.log(doSomething.prototype);
```

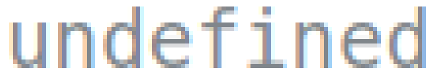


```
{
  constructor: ƒ doSomething(),
  [[Prototype]]: {
    constructor: ƒ Object(),
    hasOwnProperty: ƒ hasOwnProperty(),
    isPrototypeOf: ƒ isPrototypeOf(),
    propertyIsEnumerable: ƒ propertyIsEnumerable(),
    toLocaleString: ƒ toLocaleString(),
    toString: ƒ toString(),
    valueOf: ƒ valueOf()
  }
}
```

## Example - 2

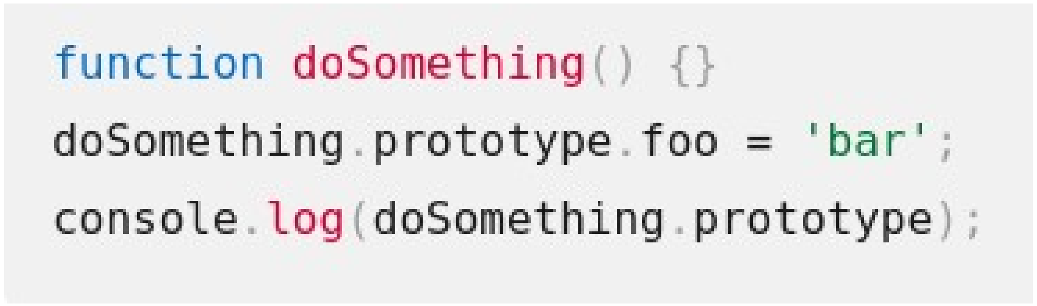


```
const doSomethingFromArrowFunction = () => {};
console.log(doSomethingFromArrowFunction.prototype);
```

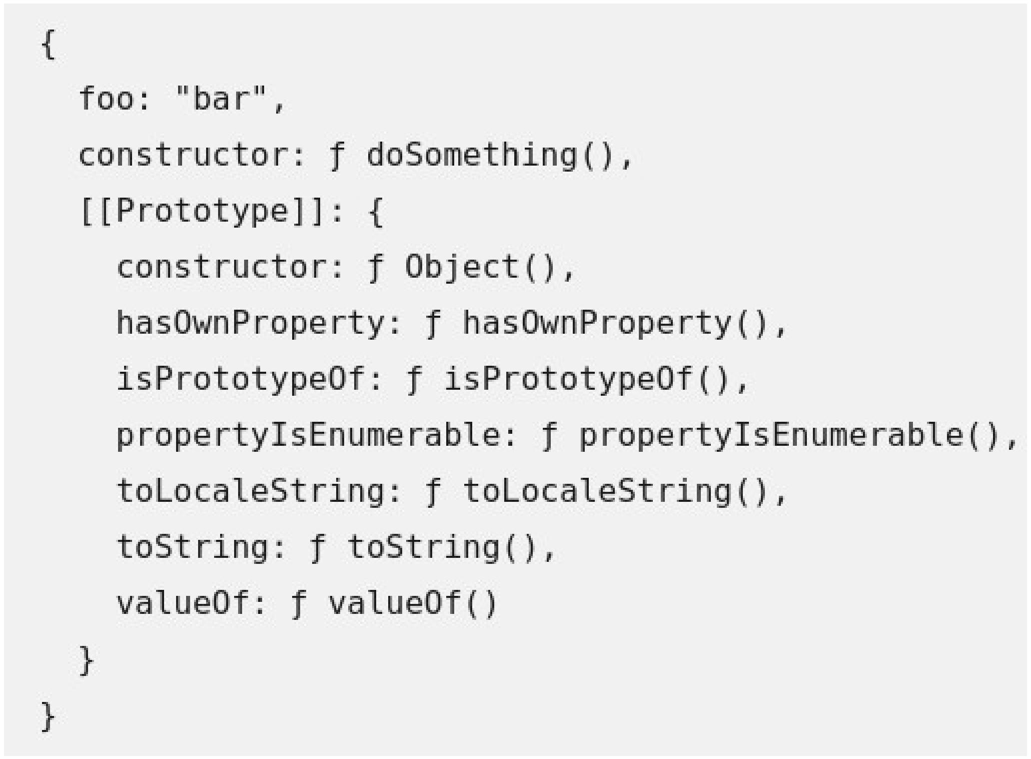


```
undefined
```

## Example - 3

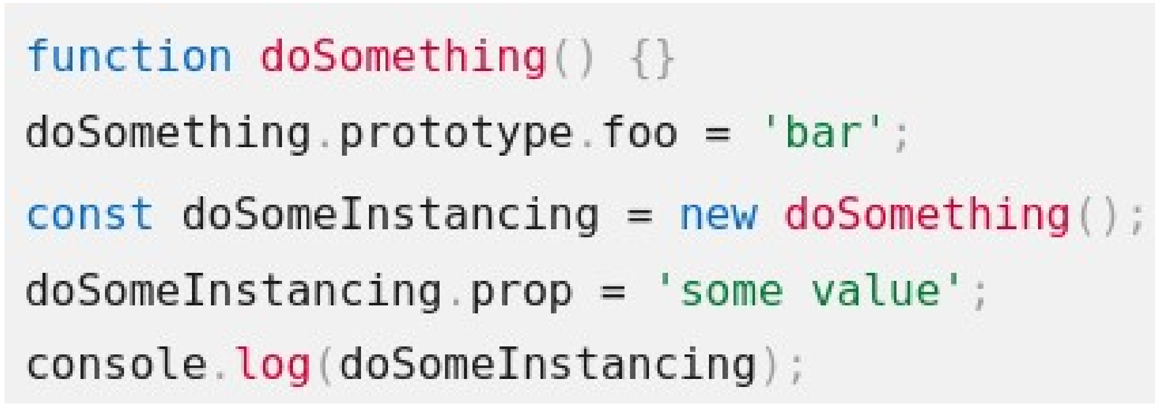


```
function doSomething() {}
doSomething.prototype.foo = 'bar';
console.log(doSomething.prototype);
```

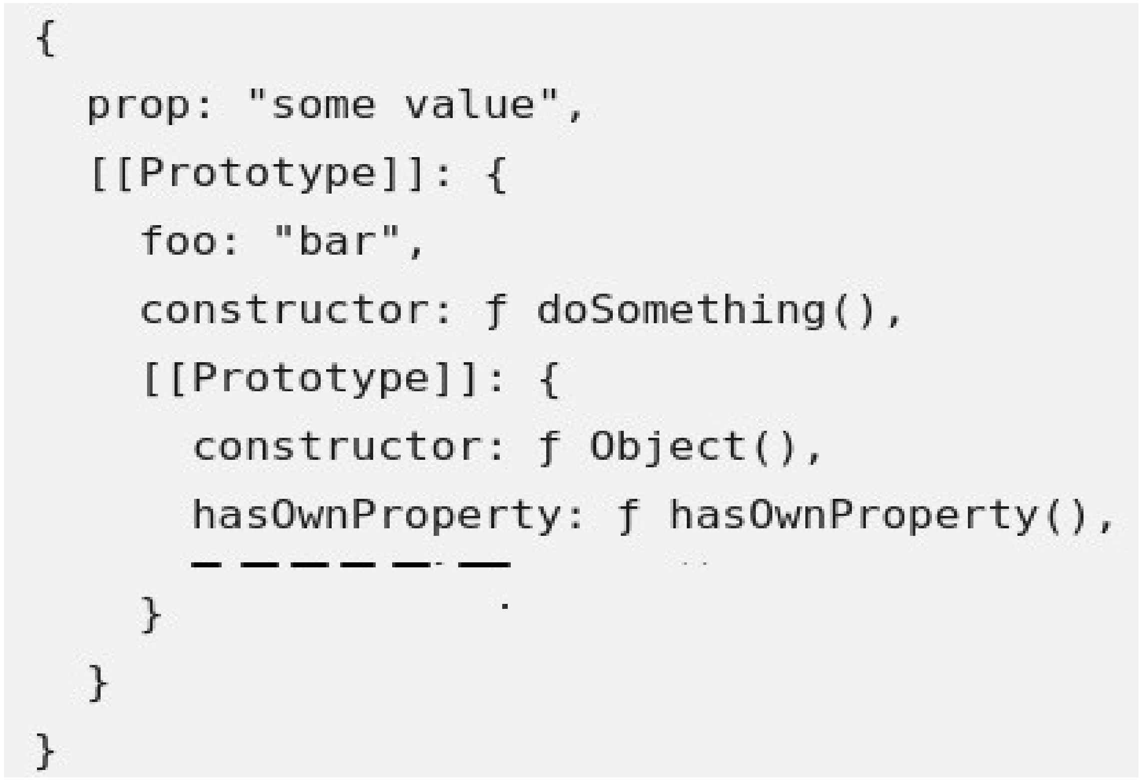


```
{
  foo: "bar",
  constructor: ƒ doSomething(),
  [[Prototype]]: {
    constructor: ƒ Object(),
    hasOwnProperty: ƒ hasOwnProperty(),
    isPrototypeOf: ƒ isPrototypeOf(),
    propertyIsEnumerable: ƒ propertyIsEnumerable(),
    toLocaleString: ƒ toLocaleString(),
    toString: ƒ toString(),
    valueOf: ƒ valueOf()
  }
}
```

# Example - 4

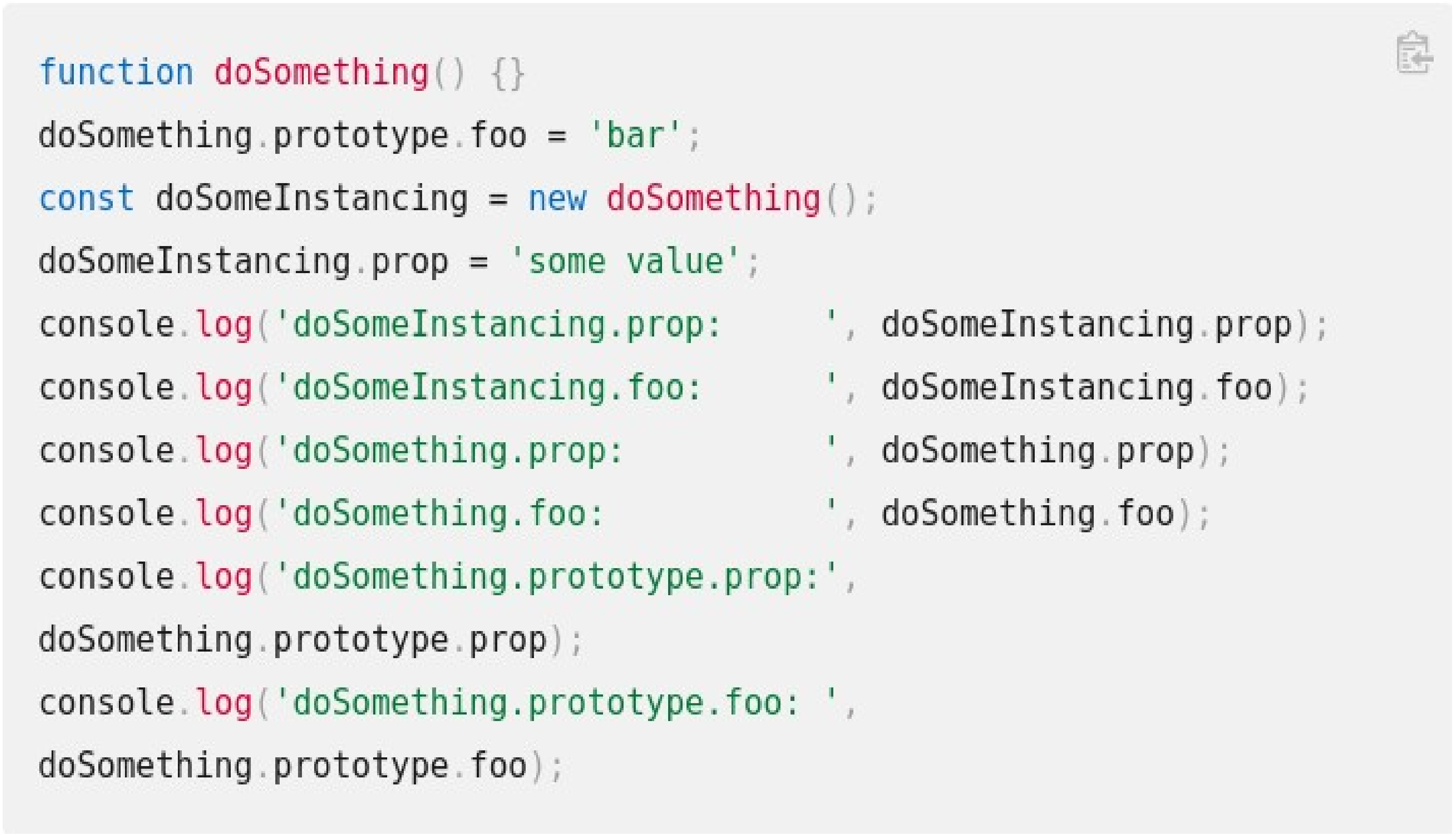


```javascript
function doSomething() {}
doSomething.prototype.foo = 'bar';
const doSomeInstancing = new doSomething();
doSomeInstancing.prop = 'some value';
console.log(doSomeInstancing);
```



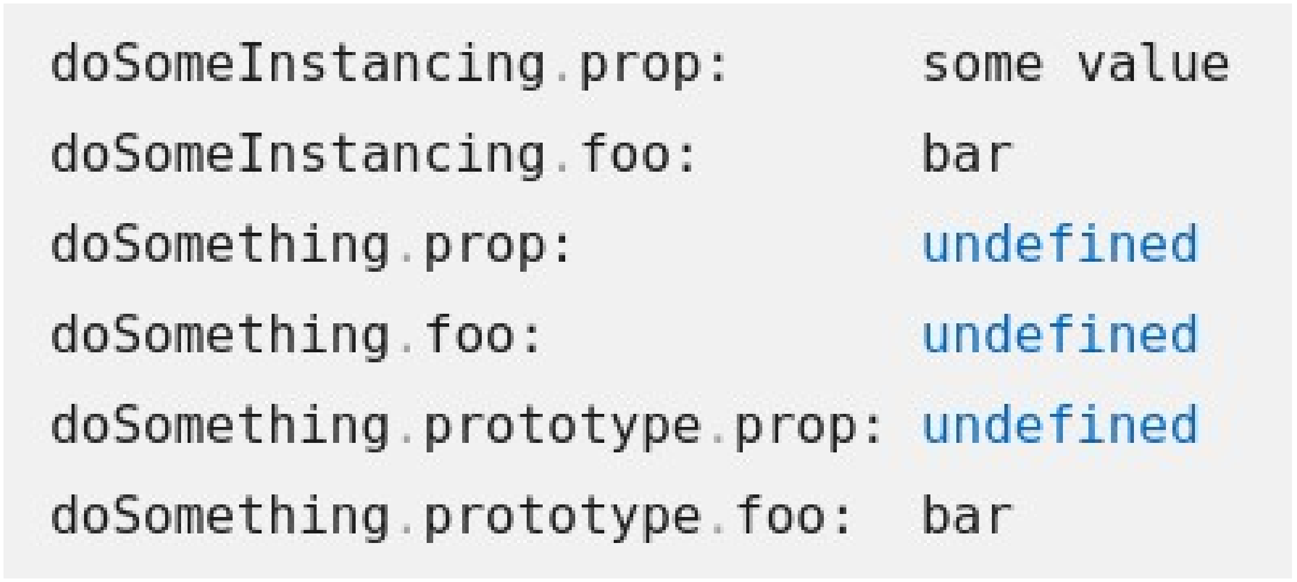
```
{
  prop: "some value",
  [[Prototype]]: {
    foo: "bar",
    constructor: ƒ doSomething(),
    [[Prototype]]: {
      constructor: ƒ Object(),
      hasOwnProperty: ƒ hasOwnProperty(),
    }
  }
}
```

# Example – 5 (Question)



```javascript
function doSomething() {}
doSomething.prototype.foo = 'bar';
const doSomeInstancing = new doSomething();
doSomeInstancing.prop = 'some value';
console.log('doSomeInstancing.prop:     ', doSomeInstancing.prop);
console.log('doSomeInstancing.foo:      ', doSomeInstancing.foo);
console.log('doSomething.prop:          ', doSomething.prop);
console.log('doSomething.foo:           ', doSomething.foo);
console.log('doSomething.prototype.prop:',
doSomething.prototype.prop);
console.log('doSomething.prototype.foo: ',
doSomething.prototype.foo);
```

# Example – 5 (Answer)



```
doSomeInstancing.prop:      some value
doSomeInstancing.foo:       bar
doSomething.prop:           undefined
doSomething.foo:            undefined
doSomething.prototype.prop: undefined
doSomething.prototype.foo:  bar
```

## Different ways to creating and mutating prototype chain

## Objects created with syntax constructs

```
const o = { a: 1 };
```

// o ---> Object.prototype ---> null

```
const b = ["yo", "whadup", "?"];
```

// b ---> Array.prototype ---> Object.prototype ---> null

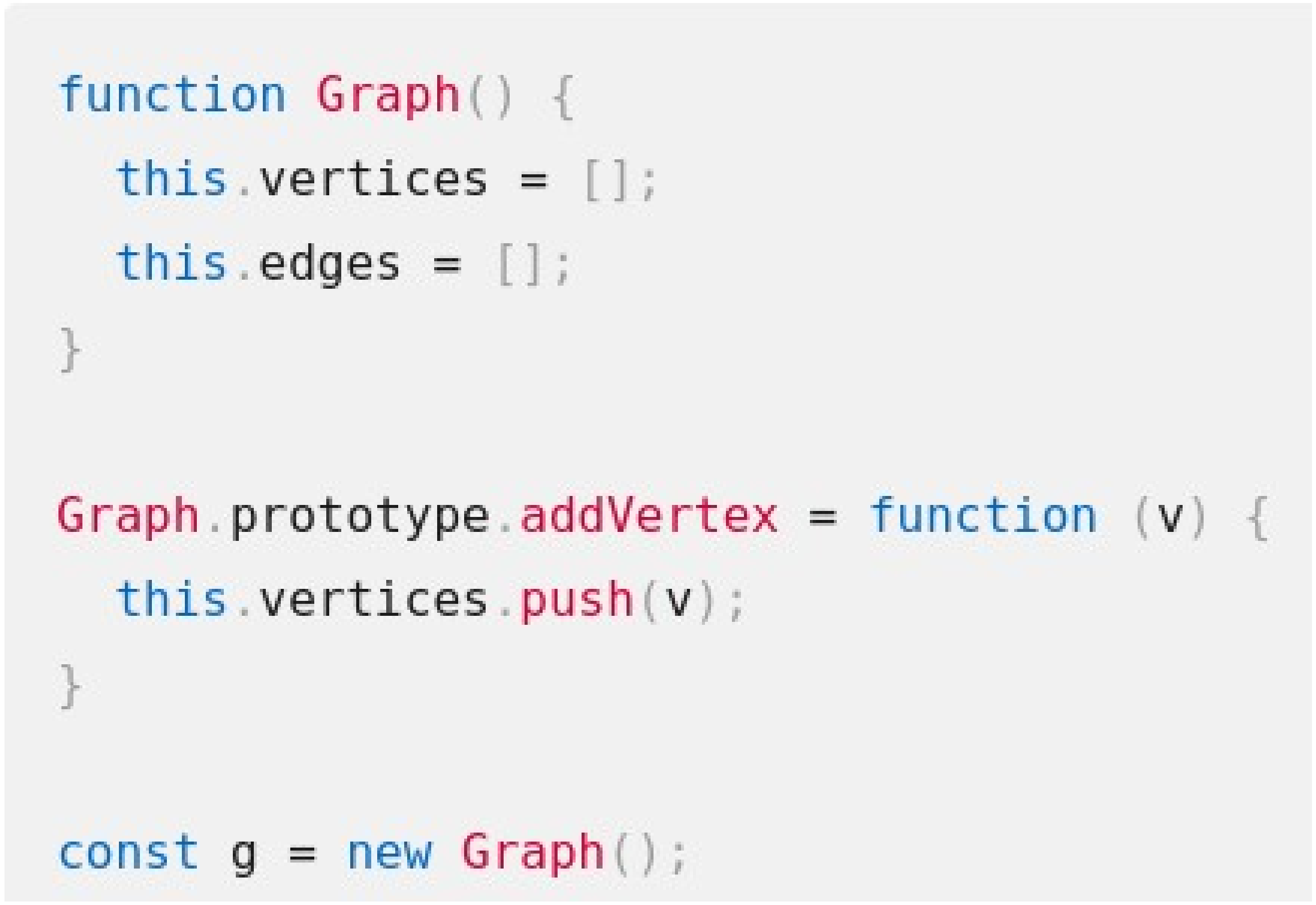
```
function f() {
  return 2;
}
```

// f ---> Function.prototype ---> Object.prototype ---> null

```
const p = { b: 2, __proto__: o };
```

// p ---> o ---> Object.prototype ---> null

# With constructor functions



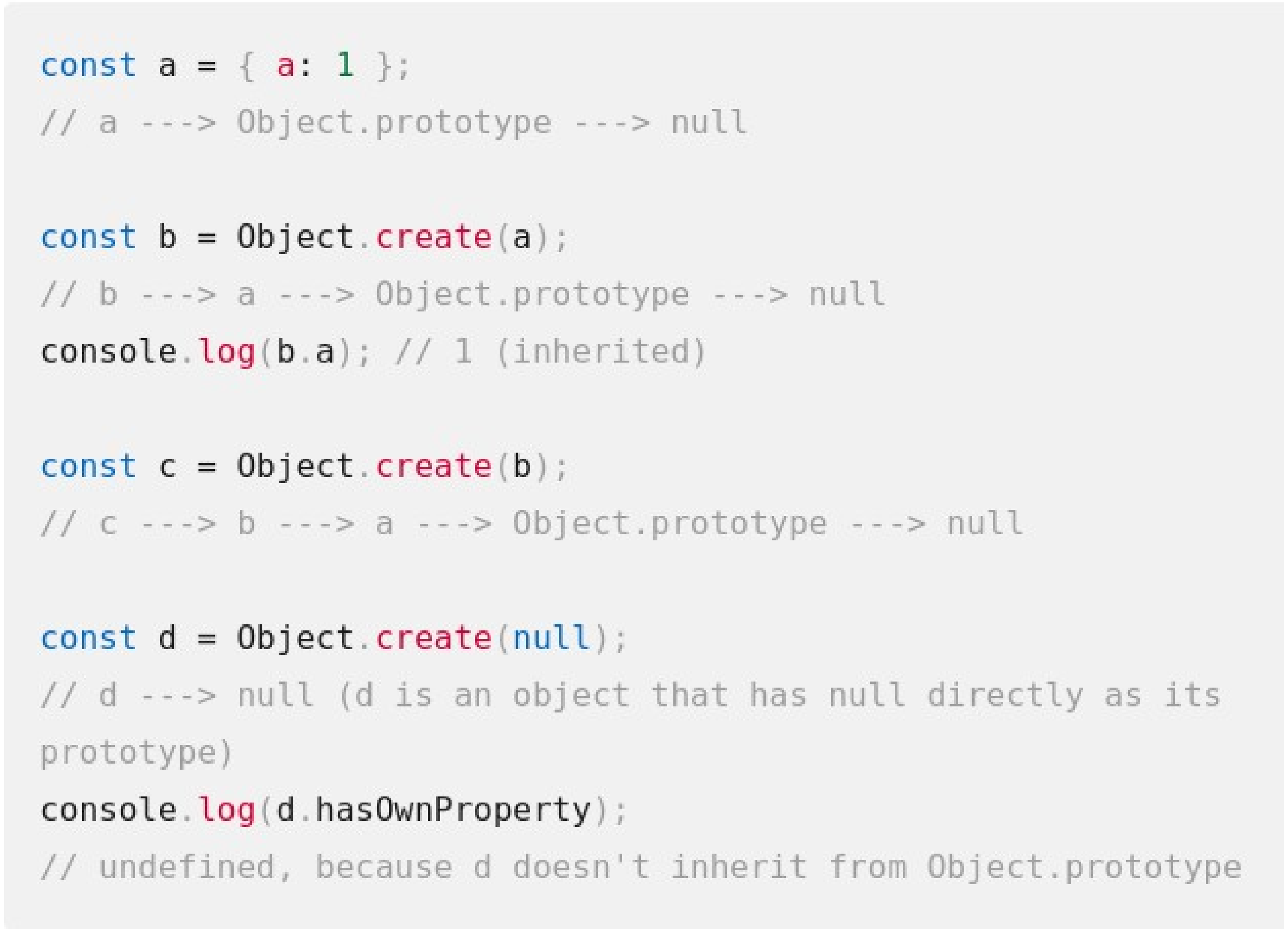
```
function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype.addVertex = function (v) {
  this.vertices.push(v);
}

const g = new Graph();
```

// g is an object with own properties 'vertices' and 'edges'.

// g.[[Prototype]] is the value of Graph.prototype when new Graph() is executed.

## With Object.create()

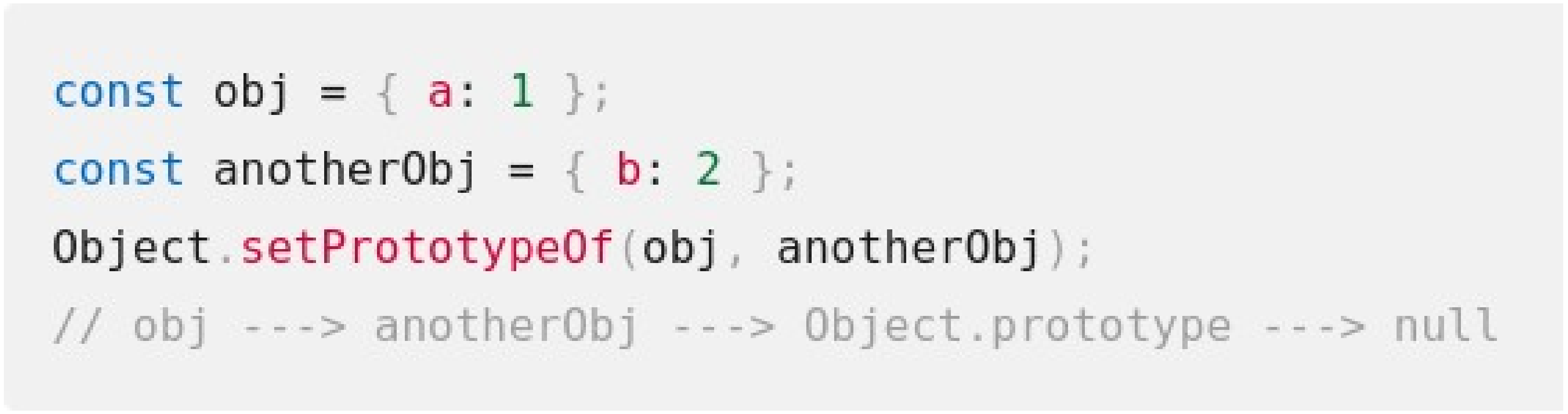
Calling **Object.create()** creates a new object. The **[[Prototype]]** of this object is the first argument of the function:



```
const a = { a: 1 };
// a ---> Object.prototype ---> null

const b = Object.create(a);
// b ---> a ---> Object.prototype ---> null
console.log(b.a); // 1 (inherited)

const c = Object.create(b);
// c ---> b ---> a ---> Object.prototype ---> null

const d = Object.create(null);
// d ---> null (d is an object that has null directly as its
prototype)
console.log(d.hasOwnProperty);
// undefined, because d doesn't inherit from Object.prototype
```

## With Object.setPrototypeOf()

While all methods above will set the prototype chain at object creation time, **Object.setPrototypeOf()** allows mutating the **[[Prototype]]** internal property of an existing object.

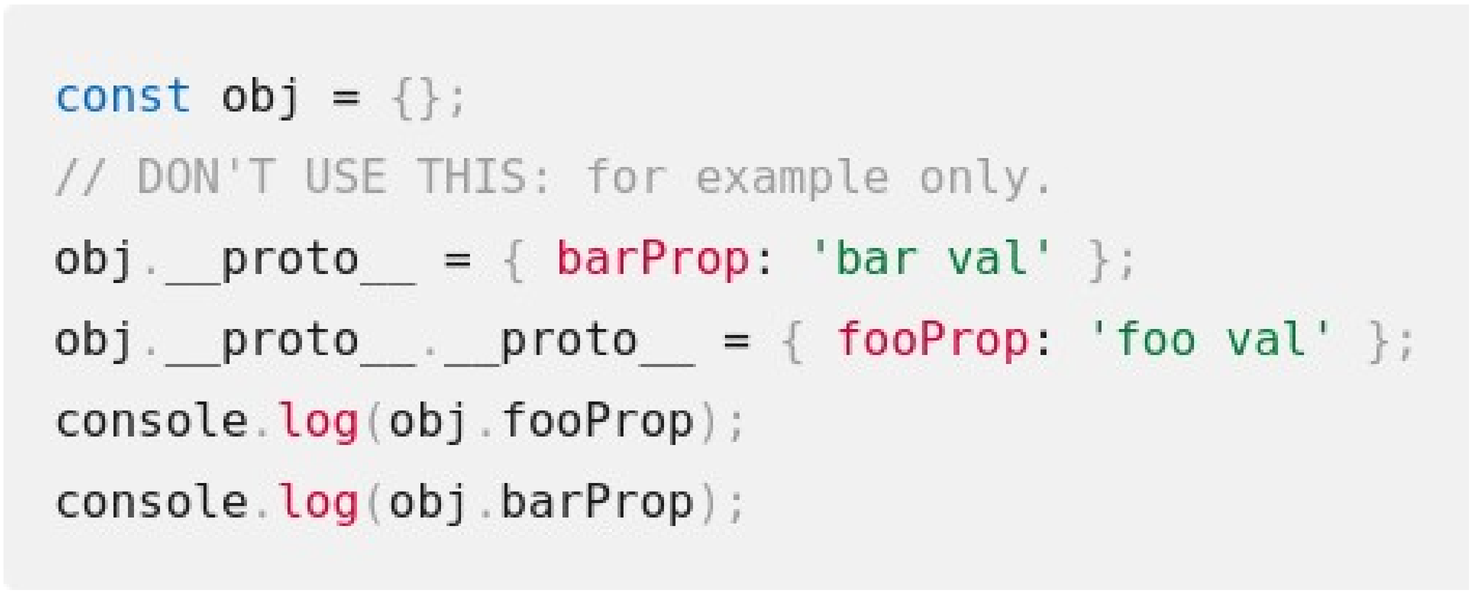


```
const obj = { a: 1 };
const anotherObj = { b: 2 };
Object.setPrototypeOf(obj, anotherObj);
// obj ---> anotherObj ---> Object.prototype ---> null
```

## With the __proto__ accessor

- All objects inherit the **Object.prototype.__proto__** setter, which can be used to set the **[[Prototype]]** of an existing object (if the __proto__ key is not **overridden** on the object).

- Object.prototype.__proto__ accessors are **non-standard** and **deprecated**. You should almost always use **Object.setPrototypeOf** instead.



```
const obj = {};
// DON'T USE THIS: for example only.
obj.__proto__ = { barProp: 'bar val' };
obj.__proto__.__proto__ = { fooProp: 'foo val' };
console.log(obj.fooProp);
console.log(obj.barProp);
```

## References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain