Javascript IV: Classes

Pau Fernández Rafa Genés Jose L. Muñoz

Universitat Politècnica de Catalunya (UPC)

Table of Contents

Prototypical Inheritance

Prototypical Inheritance

There are no classes, only objects.

There are no ”schemas” (a classe can be seen as an object ”schema”).

No ”schemas” means no *a priori* definition of how objects should be.

To define the common behavior of a group of objects, we declare that a certain object is a ”prototype” and all objects inherit its behavior.

So in this form of OOP, objects don’t belong to a class, they just have an associated prototype.

Traditional classes are then defined indirectly through the prototype, just because all objects having the same prototype behave similarly.

The Prototype Property

Every object has a `[[Prototype]]` property (which can be changed using the `__proto__` getter and setter). When we access a property on an object, and it is missing, Javascript takes it from the the prototype object.

```
let animal = {
  eat: () => console.log("nyam nyam")
}
let rabbit = {
  jump: () => console.log("Boing!"),
  type: "rabbit",
}

rabbit.__proto__ = animal
rabbit.eat()
```

The `rabbit` inherits the properties of the `animal` because the `animal` is the `rabbit`'s prototype.

The Prototype Chain

If the prototype doesn't have the property you are looking for, *keep looking for the property at the prototype's prototype, and so on.*

```
let animal = {
  walk() { console.log("A " + this.type + " is walking") },
  type: "animal",
}
let dinosaur = {
  talk() { console.log("Roar") },
  __proto__: animal,
  type: "dinosaur",
}
let tRex = {
  hunt() { console.log("Hunting you!") },
  __proto__: dinosaur,
}
tRex.walk()
```

A method is a function attached to an object. The attachment is effective in the fact that whenever we call the method on the object, the `this` variable is bound to the object.

```
let obj = {
  field: 1,
  method() {
    console.log("The field is: " + this.field)
  },
  anotherMethod() { this.field++  }
}

obj.anotherMethod()
obj.method()
```

(If we define methods with arrow functions in object literals, they get the this from the environment so it doesn't work as expected.)

Writes do not affect the prototype

When we assign to properties, these changes never affect the prototype, they affect the original object (the one before the dot in the expression).

```
let animal = {
  walk() { if (!this.isSleeping) alert(`I walk`) },
  sleep() { this.isSleeping = true }
}

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
}

rabbit.sleep()
alert(rabbit.isSleeping)  // true
alert(animal.isSleeping)  // undefined
```

Since the beginning of Javascript, creating objects involved the use of `new` with a *function constructor*:

```
function Rectangle(x, y, width, height) {
  this.x = x
  this.y = y
  this.width = width
  this.height = height
}

let r = new Rectangle(0, 0, 80, 45)
```

Javascript creates a new, empty object, that is bound to the `this` variable during the function constructor's execution, and gives that object as result.

The **prototype** Property

But the function constructor is an object (a Function object), and it has a special field called **prototype** that is *assigned to new objects as their prototype*:

```
let figure = {
  area() { return this.width * this.height }
}
function Rectangle(x, y, width, height) {
  this.x = x
  this.y = y
  this.width = width
  this.height = height
}
Rectangle.prototype = figure

let r = new Rectangle(0, 0, 80, 45)
console.log(r.area())
```

The Default Prototype

If we do not change the **prototype** property of a function constructor, it will have one like this:

```
function Rectangle(x, y, width, height) {
  ...
}
Rectangle.prototype = { constructor: Rectangle }
```

The **constructor** property of the prototype allows us to, given an object, construct another one using the original constructor:

```
let r = new Rectangle(0, 0, 70, 30)
let r2 = new r.constructor(10, 10, 40, 20)  // new Rectangle
```

However, when we change the **prototype** property of **Rectangle** (as we did with the **figure**), we lose the **constructor** property, which was set by Javascript.

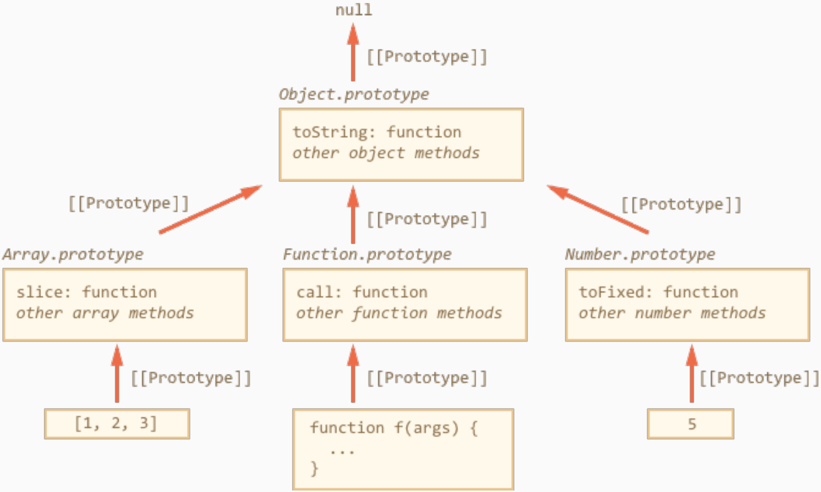
Adding Stuff to the Prototype

In general, it is better to add functions or fields to the prototype, instead of replacing it altogether:

```
function Circle(x, y, radius) {
  this.x = x
  this.y = y
  this.radius = radius
}
Circle.prototype.area = function() {
  return 2 * Math.PI * this.radius
}
```

In this way, we don't lose the `constructor` property.

Native prototypes



Native prototypes proof

We can check the prototypes of basic objects along the prototype chain:

```
let arr = [1, 2, 3]

// it inherits from Array.prototype?
alert(arr.__proto__ === Array.prototype)  // true

// then from Object.prototype?
alert(arr.__proto__.__proto__ === Object.prototype)  // true

// and null on the top.
alert(arr.__proto__.__proto__.__proto__)  // null
```

Changing Native Prototypes and Polyfills

Javascript will let you change native prototypes (!):

```
String.prototype.show = () => { console.log(this) }
"OMG!".show()
```

It is generally a bad idea to do this, because it is easy to produce conflicts in code using many libraries.

One case is very important, though: **polyfills**. A polyfill is a piece of code that patches a Javascript implementation which is lacking a specific method or group of methods. For instance:

```
if (!String.prototype.repeat) { // if there's no such method
  String.prototype.repeat = function(n) {
    return new Array(n + 1).join(this)  // naïve implementation!
  }
}
```

Let's suppose that we have two classes Person and Superhero, and we want to make all Superheros also Persons.

```
function Person(name) {
  this.name = name
}
Person.prototype.sayHi = function() {
  console.log("Hi, I'm " + this.name)
}

function Superhero(name, hero) {
  this.name = name
  this.hero = hero
}
Superhero.prototype.breakThroughWall = function() {
  console.log("Look! " + this.hero + " broke through a wall!")
}
```

We want that every new object constructed with Superhero has as prototype the object `Superhero.prototype`, but also that this prototype has itself a prototype which is `Person.prototype`.

To express that any `Superhero` is also a `Person` we just need to connect both prototypes:

```
Superhero.prototype.__proto__ = Person.prototype
```

Now we can write:

```
let bob = new Superhero("Bob Parr", "Mr. Incredible")
bob.sayHi()
bob.breakThroughWall()
```

Table of Contents

ES6 Classes

The New Class Syntax

In ES6, a new syntax was developed, closer to the syntax of other OOP languages:

Classes are defined with the `class` keyword:

```
class Person {
  constructor(name) {
    this.name = name
  }
  sayHi() {
    console.log("Hi, I'm " + this.name)
  }
}
```

The `constructor` is now a method.

Inside a class declaration there can't be any "`field: value`" assignments, only methods.

Derived Classes with **extends**

Derived classes are defined with the **extends** keyword (from Java again):

```
class Superhero extends Person {
  constructor(name, hero) {
    super(name)
    this.hero = hero
  }
  breakThroughWall() {
    console.log("Look! " + this.hero + " broke through a wall!")
  }
}
```

The **super** keyword allows us to call methods in the super-class (or base class).

You have to call **super** first in your constructor

If a class doesn't have a constructor, one with the following signature is provided:

```
class X {
  constructor(...args) { // Default constructor (if not present)
    super(...args)
  }
}
```

You can't omit the `super()` call in a derived class constructor:

```
class Superhero extends Person {
  constructor(hero) {
    this.hero = hero
  }
}
let bob = new Superhero("Mr. Incredible")  // error: this is not defined.
```

You must call the `super` constructor, and do so *before* doing anything else (Javascript requirement).

Getters and Setters

You can define methods to **get** and **set** a fictitious field:

```
class Person {
  constructor(name) {
    this.privateName = name
  }
  get name() { return this.privateName }
  set name(newname) { this.privateName = newname }
}
let p = new Person("Bob")
console.log(p.name)
p.name = "Roberta"
console.log(p.name)
```

We might think there is a field called `name`, but the real field containing the name is `privateName`, and the `get` and `set` methods emulate the assignment and access to the `name` "field".

Static Methods

Static methods are class methods, not associated with particular objects but with the class itself. The old way of achieving this would be directly assigning a new field of the constructor (which we do with `prototype`):

```
function User(name) { this.name = name }
User.staticMethod = function() {
  console.log("Hello from User.staticMethod!")
}
```

The same code in the new class syntax with the `static` keyword:

```
class User {
  constructor(name) { this.name = name }
  static staticMethod() {
    console.log("Hello from User.staticMethod!")
  }
}
```

Table of Contents

Exceptions

Stack Traces

Errors in Javascript ...

```
function second(obj) {
  obj.clearlyNonExistentMethod()
}
function first() {
  let x = { a: 1, b: "2", c: [3] }
  second(x)
}
first()
```

... usually end up showing a stack trace (the exact functions that were in the stack waiting when the error was produced):

```
TypeError: obj.clearlyNonExistentMethod is not a function
  at second (/home/pauek/.../stack-trace.js:2:7)
  at first (/home/pauek/.../stack-trace.js:6:3)
  at Object.<anonymous> (/home/pauek/.../stack-trace.js:8:1)
  ...
```

To catch errors, we use a **try**/**catch** block.

```
try {
  // Code that could have errors
} catch (e) {
  // Error handling code
}
```

The **try** code includes any instructions that we know can produce errors.

The errors will be catched even if they are produced by functions called *at a different depth* in the call stack.

The catch clause catches any errors (of any type) ocurred within the **try** block.

When an error is produced within the **try** block, the execution point jumps directly to the **catch** clause, skipping any pending code in the **try** block.

Catching Error Example

The `catch` block receives an `Error` object which describes the error.

```
function second(obj) {
  obj.clearlyNonExistentMethod()
}
function first() {
  let x = { a: 1, b: "2", c: [3] }
  second(x)
}
try {
  first()
} catch (e) {
  console.log("Error Type: ", e.name)
  console.log("Error Message: ", e.message)
}
```

Throwing Errors

Errors can be produced with `throw` passing an `Error` object.

```
throw new Error("I just bit my tongue!")
```

The error can be any object or value but in general it is good practice to at least have the two fields `name` and `message`.

There exist already many types of errors defined in Javascript:

```
let err1 = new SyntaxError("Just trolling, muahaha")
let err2 = new ReferenceError("This one is legit, though")
```

You can of course define your own `Error` classes:

```
class EmbarrassingError extends Error {
  constructor(message) {
    super(message)
    this.name = "EmbarrassingError"
  }
}
```

Discriminating Errors

To discriminate errors, the `name` field in the `Error` object indicates the type:

```
try {
  // Error prone code...
} catch (e) {
  if (e.name === "EmbarrassingError") {
    console.log("Everything is cool, actually...")
  } else if (e.name === "TypeError") {
    // ...
  } else {
    throw e   // Rethrow for other try-catch blocks!
  }
}
```

After handling the errors you can address, you should throw the error for other `try`-`catch` blocks lower in the stack.

Table of Contents

Maps

Maps

An `Object` is a dictionary (or table) of key-value pairs, in which keys are *strings*.

A `Map` is a also a dictionary of key-value pairs, but keys can be **of any type**.

Main methods:

- `new Map()` — creates a map,
- `map.set(key, value)` — stores a value by the `key`,
- `map.get(key)` — searches by `key`, returns `undefined` if not found,
- `map.has(key)` — returns if `key` is in `map`,
- `map.delete(key)` — erases key-value pair,
- `map.forEach(func)` — calls `func` for every key-value pair,
- `map.clear()` — removes all pairs from the map,
- `map.size` — returns the element count.

An Example

```
let map = new Map()

map.set('1', 'str1')        // a string key
map.set(1, 'num1')          // a numeric key
map.set(true, 'bool1')      // a boolean key

// Map keeps the type, so these are different
console.log( map.get(1)   )  // 'num1'
console.log( map.get('1') )  // 'str1'

console.log( map.size )      // 3
```

Objects as Keys

If you want to store a visit counter for each user without `Map`s, you would probably do:

```
// Need the 'id' since many users might have the same name...
let user = { name: "John", id: 42 }
let visitCounts = {}
visitCounts[user.id] = 7
console.log(visitCounts[user.id])
```

The solution with `Map`s is much more elegant, because we can use the `user` object itself as key:

```
let user = { name: "John" }
let visitCounts = new Map()
visitCounts.set(user, 7)
console.log(visitCounts.get(user))
```

How a `Map` Compares Keys

Maps, internally, need to know if two keys are different.

The algorithm for this is called `SameValueZero`:

https://tc39.github.io/ecma262/#sec-samevaluezero

This algorithm is almost the same as the triple-equals comparison, except for one case:

- When comparing `NaN` with `NaN`, the `===` operator returns `false` but a Map considers this comparison `true`. [1]

(This algorithm can't be changed or customized.)

[1] This makes sense since there should be only one item associated with the key `NaN`.

Chaining in `Map.prototype.set`

Every `Map.prototype.set` call returns the map itself, so you can chain them.

This code:

```
let map = new Map()
map.set('1', 'str1')
map.set(1, 'num1')
map.set(true, 'bool1')
```

is equivalent to this code:

```
let map = new Map()
map.set('1', 'str1')
   .set(1, 'num1')
   .set(true, 'bool1')
```

Map Initialization

To initialize a map, we can use an array of arrays with key value pairs:

```
let map = new Map([
    ['1', 'str1'],
    [1, 'num1'],
    [true, 'bool1']
])
```

We cannot initialize a Map from an object, but we can use `Object.entries` to produce an array of key-value pairs for the object properties:

```
let map = new Map(Object.entries({
    foo: 'bar',
    bar: 'baz',
}))
```

Iterating a Map: using keys

The `Map.prototype.keys` method returns an iterable with the keys of a map:

```
let recipeMap = new Map([
    ['cucumber', 500],
    ['tomatoes', 350],
    ['onion',    50]
])

let sum = 0
for (let vegetable of recipeMap.keys()) {
    sum += recipeMap.get(vegetable)
    console.log(vegetable)// cucumber, tomatoes, onion
}
console.log(sum)// 900
```

(Interestingly, the map remembers the insertion order and iterations occur in that order.)

Iterating a Map: using only values

If we just want to sum the values we can use the `Map.prototype.values` methods, which returns an iterable for values:

```
let recipeMap = new Map([
    ['cucumber', 500],
    ['tomatoes', 350],
    ['onion',    50]
])

let sum = 0
for (let value of recipeMap.values()) {
    sum += value
}
console.log(sum)// 900
```

Iterating a Map: using pairs

By either using `Map.prototype.entries` or leaving the `for...of` iteration as is, we iterate over key-value pairs:

```
let recipeMap = new Map([
    ['cucumber', 500],
    ['tomatoes', 350],
    ['onion',    50]
])

for (let pair of recipeMap) { // also with recipeMap.entries()
    let key = pair[0]
    let value = pair[1]
    console.log(key, value)
}
```

Iterating a Map: using **forEach**

A `Map.prototype.forEach` method exists, which receives a function which will be called with the following parameters: value, key, and the map itself.

```
let recipeMap = new Map([
    ['cucumber', 500],
    ['tomatoes', 350],
    ['onion',    50]
])

recipeMap.forEach((value, key, map) => {
    console.log(`key = ${key}, value = ${value}`)
})
```

Table of Contents

Sets

Sets

A set is a collection of values in which each element can appear only once.

Main methods:

- `new Set(iterable)` — creates a set,
- `set.add(value)` — adds a new `value` to the set,
- `set.delete(value)` — deletes a `value` from the set,
- `set.has(value)` — returns `true` if `value` is in the set,
- `set.clear()` — removes all elements from the set,
- `set.size` — returns the element count.

An Example

```
let visitors = new Set()

let john = { name: "John" }
let pete = { name: "Pete" }
let mary = { name: "Mary" }

visitors.add(john)
visitors.add(pete)
visitors.add(mary)
visitors.add(john)
visitors.add(mary)

console.log( visitors.size )// 3

for (let user of visitors) {
    console.log(user.name)// John (then Pete and Mary)
}
```

Iterating a Set

Iterating a set can be accomplished with the `for...of` syntax:

```
let fruits = new Set(["oranges", "apples", "bananas"])
for (let name of fruits) {
    console.log(name)
}
```

There is also a `Set.prototype.forEach` method:

```
fruits.forEach((value, valueAgain, fruits) => {
    console.log(value)
})
```

Notice how the function **receives the `value` 2 times**. This is for compatibility with map, so that a Map can be exchanged for a Set and not having to rewrite `forEach` callbacks.

For the same reason, Sets also support `keys`, `values` and `entries` even if they don't make much sense in Sets.

Garbage Collection Revisited

In general, objects are garbage collected when you loose the last reference you have to them.

```
let obj = {
    name: "John Doe",
    age: 27,
}
obj = null// the object is garbage collected
```

But if you put them into a Map as keys, they are not garbage collected as long as you have a reference to the map itself:

```
let map = new Map()
let obj = {
    name: "Jane Doe",
    age: 42,
}
map.set(obj, "abcde")
obj = null// NOT garbage collected because it is referenced by the map
```

WeakMaps

`WeakMap`s are similar to `Map`s but with some crucial differences:

- They can only have objects as keys.
- Whenever this objects are garbage collected they are removed from the WeakMap ***automatically***.
- We can't iterate them (no `keys`, `values` or `entries` methods).

WeakMaps methods are only:

- `weakmap.set(key, value)`
- `weakmap.get(key)`
- `weakmap.has(key)`
- `weakmap.delete(key)`

Limitations in the iteration and size of the map come from the inherent unpredictability of the Garbage Collector, which indirectly determines the size of the WeakMap, and what elements it still has at any given instant.

Why do we need WeakMaps?

WeakMaps are useful if you want to associate a piece of data to an object but you need to store that data not on the object itself but somewhere else.

By putting the associated data in the WeakMap you have this data available while the object lives, but do not prevent the system from garbage collecting the object if necessary (which would happen with a normal map).

```
// Store visit counts for users
let user = {
    name: "John Doe",
    age: 27,
}
let visitCounts = new WeakMap()
visitCounts.set(user, 2)

user = null// "John Doe" will be gargabe collected

// Even if we don't know exactly when, the visit count associated with
// "John Doe" will be removed from the map.
```

`WeakSet`s are to `Set`s what `WeakMap`s are to `Map`s. So they are similar to sets with these constraints:

- We can only add objects to `WeakSet`s.
- Whenever an object is garbage collected it is removed from the `WeakSet`.
- They do not have iteration or size methods (only `add`, `has` and `delete`).

It is useful to think of a `WeakSet` as a `WeakMap` that only lets us associate a Boolean value with an object. It is an external ”tag” for an object.

WeakSets to validate objects

In this example, to prohibit usage of badly initialized objects of type ApiRequest the class keeps a record of objects that have gone through the constructor (so they are properly initialized).

```
// Keep a set of validly initialized request objects
const requests = new WeakSet()

class ApiRequest {
    constructor() {
        // Initialize properly
        requests.add(this)// "Tag" an object as valid
    }

    makeRequest() {
        if (!request.has(this)) {
            throw new Error("Invalid access")
        }
        // Do the request
    }
}
```

Table of Contents

The `Date` class

The `Date` class

`Date` is a builtin object. It stores date and time, and provides methods for managing its value.

Dates have:

- A variety of constructors to create a date from the computer clock, a timestamp, a string, etc.
- Methods to get various parts: year, month, day, hour, minutes, etc.
- Methods to set those parts.
- A method to parse a Date from a string (`Date.parse`).
- A method to obtain the current timestamp (`Date.now`).

Methods to set the various parts of a date ”auto-correct” the values you pass, so you can easily compute time displacements of any size.

Creating a **Date**

With the current date and time:

```
let now = new Date()
console.log(now)
```

With a number of milliseconds (the **timestamp**), which represents the milliseconds that have passed since January 1st, 1970:

```
let beginningOfTime = new Date(0)
console.log(beginningOfTime)  // 1970-01-01T00:00:00.000Z
```

With a textual Date as a string (parsed with `Date.parse`):

```
let birthday = new Date("1950-10-06")
console.log(birthday)
```

Creating a **Date** with explicit values

A constructor lets us create a day from its 7 components:

```
new Date(year, month, day, hours, minutes, seconds, milliseconds)
```

When creating a `Date` this way, keep in mind that:

- The `year` must have 4 digits. (98 not allowed.)
- The `month` starts at 0 (which is January).
- If the `day` is absent, it is assumed to be 1 (which is the first day of the month).
- If the `hours`, `minutes`, etc. are absent, they are assumed to be 0.

```
let kingsday = new Date(2019, 0, 6)
```

Getting `Date` components

There are methods for all of `Date`'s components:

- `getFullYear` — get the year.
- `getMonth` — get the month, **from 0 to 11**.
- `getDate` — get the day (of the month).
- `getDay` — get the day (of the week, sunday is 0).
- `getHours` — get the hours.
- `getMinutes` — get the minutes.
- `getSeconds` — get the seconds.
- `getMilliseconds` — get the milliseconds.

Timezones

Date components are relative to your local time zone, there are equilvalent methods for UTC converted components:

- `getUTCFullYear`
- `getUTCMonth`
- `getUTCDate`
- `getUTCDay`
- `getUTCHours`
- `getUTCMinutes`
- `getUTCSeconds`
- `getUTCMilliseconds`

```
let now = new Date()
console.log(now.getHours())  // 10
console.log(now.getUTCHours())  // 9
```

Setting date components

Analogous methods let you set date components:

- `setFullYear(year[, month, date])`
- `setMonth(month[, date])`
- `setDate(date)`
- `setDay(day)`
- `setHours(hour[, min, sec, ms])`
- `setMinutes(min[, sec, ms])`
- `setSeconds(sec[, ms])`
- `setMilliseconds(ms)`
- `setTime(timestamp)` — set the timestamp directly.

Autocorrection

If you pass values to the set methods that exceed the limits, they are computed as the correct date. For instance:

```
let Feb1st = new Date(2019, 0, 32)
console.log(Feb1st)  // 2019-02-01T00:00:00.000Z

let May1st = new Date(2018, 4, 2)
May1st.setDate(0)    // first day of the month is 1,
                     // 0 is the last day of the previous month...
console.log(May1st)  // 2018-04-30T08:00:00.000Z
```

This is true for any of the components, so in this way you can compute time differences:

```
let nextWeek = new Date()
// next week
nextWeek.setDate(now.getDate() + 7)
console.log(nextWeek)
```

Dates as timestamps

Since dates are stored internally as a timestamp, several things are straightforward since dates can be seen as a timestamp in several contexts.

Copying a `Date` is like copying the timestamp:

```
const copyDate = (d) => new Date(d)  // d -> timestamp
```

Date comparison just uses the timestamp as an integer:

```
const isInThePast = (d) => {
    let now = new Date()
    return d < now
}
```

Subtracting two dates just subtracts the timestamps:

```
let now = new Date()
let Feb1st = new Date(2018, 1, 1)
console.log("Difference in ms: " + (now - Feb1st))
```

Date.now

If we just need the timestamp, it is usually better to avoid the creation of a `Date` object by using `Date.now` (especially important in benchmarking):

```
let dstart = new Date()
for (let i = 0; i < 1000000; i++) {
    // work, work, work...
}
let dend = new Date()
console.log('The loop takes ' + (dend - dstart) + ' milliseconds')
```

By using `Date.now`, no new objects are created and therefore no time is spent garbage collecting:

```
let tstart = Date.now()
for (let i = 0; i < 1000000; i++) {
    // work, work, work...
}
let tend = Date.now()
console.log('The loop takes ' + (tend - tstart) + ' milliseconds')
```