Cloud Computing (INGI2145) - Lab Session 5

Nicolas Laurent, Marco Canini

Introduction to Javascript

Today we'll learn the basics of the Javascript language as well as Node.js, a Javascript backend that runs on a server instead of in the browser.

The only thing you need to do to follow this session is install Node.js from the official website (http://nodejs.org/) .

Old versions of the course's VM are misconfigured. To fix the issue, run the following commands:

```
sudo aptitude remove node
sudo ln -s $(which nodejs) /usr/local/bin/node
```

What is Javascript?

Javascript is mostly famous for being the only widely adopted client-side web language: websites can include scripts that will be run within the browser. This enables the page to create dynamically computed data instead of a page that was generated on the server-side.

With the advent of Node.js, Javascript is now making forays on the server side as well, as people realize that using the same language on the client side and the server side has some advantages (see the meteor framework (https://www.meteor.com/) for an extreme example of this).

Technically speaking, Javascript is a dynamically-typed, object-oriented, event-driven scripting language with first-class functions. Unlike most object-oriented languages, Javascript does not revolve around classes (although it tries very hard to pretend it does), but around prototypes. During this session, we will spend quite a bit of time on that distinction.

You're encouraged to pay attention during this session, as you'll have to write Javascript for the second assignment!

We will use the ECMA 5.1 Javascript standard, as it is what is used by Node.js and is compatible with all modern browsers.

An Overview of Javascript

First, open a Javascript read-eval-print loop (the "REPL") by typing `node` in the command line. You can type Javascript expressions and statements in there. When you validate (with enter), Node will evaluate the code and print the result. It also applies the side-effects (such as assignments) to the environment shared by all the code that you enter in the REPL. To reset this environment, exit the REPL (by typing CTRL-C twice) then restart it. You may find the `console.log(value)` call useful.

There are **a lot** of Javascript tutorials and introductions on the Internet. Instead of reinventing the wheel, we will use them to get a general idea about the language.

Read one or both of the following:

- Read Learn Javascript in Y Minutes (http://learnxinyminutes.com/docs/javascript/) . Read until the line that reads `var MyConstructor = function() {` .
- Read A re-introduction to JavaScript (https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript) . Read until the "Custom objects" section (non-inclusive).

Both have significant overlap, so pick what seems most appealing to you. The first option is more dense while the second one has more extensive examples.

We encourage you to type some of the examples in the REPL, and to explore a bit; as well as to ask any questions you might have to me (the friendly TA) or Google (the evil search overlord).

You should plan to not spend more than 30 minutes on these tutorials.

Prototypal Inheritance

Javascript is pretty minimalistic, the only types it knows about are:

- numbers (64-bit floating point)
- strings
- booleans
- functions
- objects

You can get the type of a value by using the `typeof` operator.

```
typeof 1 // 'number'
```

Let’s focus on objects. Objects are really dictionaries with string keys. Javascript calls its keys “properties”. If the properties are also valid identifiers, you can access their associated values using the dot notation (e.g., `x.my_boring_key_1`). Otherwise, you need to use the bracket notation (e.g., `x['(>'-')>']`). You can delete properties by using the `delete` operator: `delete x.my_boring_key_1;`.

Each object also has a parent dictionary, called its prototype. The portable way to access the prototype of an object `x` is to call `Object.getPrototypeOf(x)`. However, most implementation define the special property `__proto__` to access the prototype. You can also use `__proto__` to set the prototype of an object, although, again, that is not really standard.

When you try to access a property of an object, Javascript checks if the object itself has a property with the given name. If it has, it retrieves its value. Otherwise, it checks the object's prototype. If the prototype does not have the property, its own prototype (the prototype's prototype) is checked, and so on until a `null` prototype is encountered. In other words, we walk the "prototype chain".

To create an object with a given prototype, use `Object.create(prototype)`. The object will be dynamically linked to the prototype: if the value of a prototype's property changes, it changes for the object as well, unless the object has overridden the value.

```
proto = {x: 1, y:2}
obj = Object.create(proto)
obj.x // 1
proto.x = 2
obj.x // 2
obj.y = 3
proto.y = 4
obj.y // 3
```

Note that properties can be functions! In that case, when calling the function on the object, the special variable `this` will be set to the object inside the function.

```
x = {id: 42, f: function() { return this.id }}
x.f() // returns 42
```

You can change what `this` refers to by using the `call` function:

```
x.f.call({id: 52}) // returns 52
```

Classes in Javascript

The prototypal model is simple enough. Unfortunately, it was made more complicated by trying to emulate classes with prototypes.

While we encourage you to stick to prototypes, you will encounter code that uses classes. In fact, the language's base API uses them extensively.

First, we remark that speaking of classes is kind of a stretch. Strictly speaking, Javascript has constructors. Internally however, Javascript has an attribute called `[[Class]]` which it equates with the constructor. There are ways (http://perfectionkills.com/instanceof-considered-harmful-or-how-to-write-a-robust-isarray/) to access it however.

So Javascript has constructors. A constructor is a function that initializes a new object. Inside a constructor, `this` refers to a newly created object, whose prototype is `constructor.prototype`.

A few fine points. First, functions are a sub-type of object. This means they can have arbitrary properties, so `constructor.prototype` is not an aberration.

Second, don't confuse `constructor.prototype` with `constructor.__proto__`! The `prototype` property is the prototype we want the constructed object to have, it is not the constructor's own prototype.

Finally, when we use this pattern we usually make `constructor.prototype.constructor` refer back to `constructor`. Some API functions may depend on this.

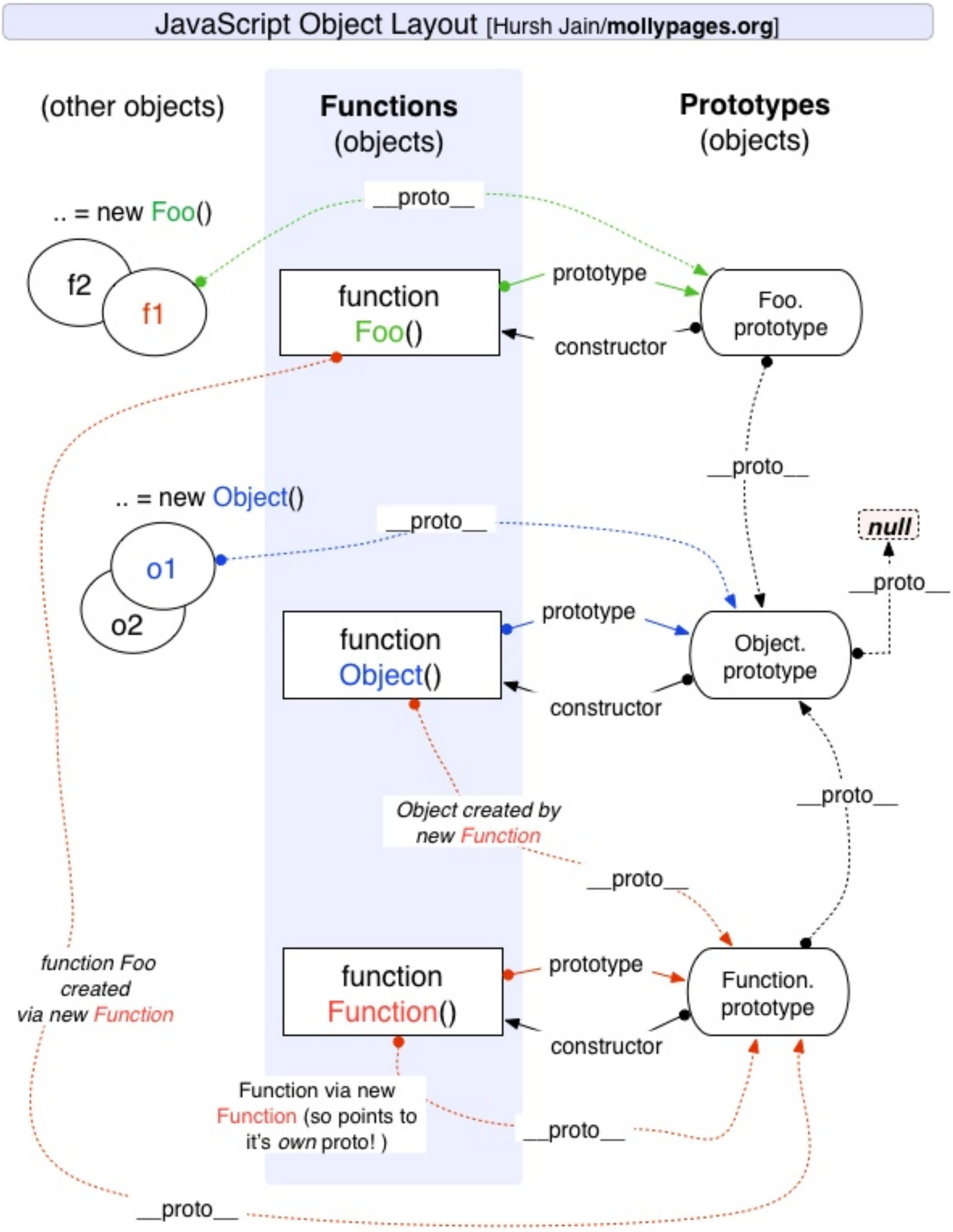
If a constructor returns a value, then that value will be used instead of the value passed into the constructor, which can be a cute trick sometimes.

Last but not least: since constructors are just functions, how do we tell Javascript to treat them specially (to populate `this` with a new object, then return that or whatever the function returns)? The answer is to use the `new` operator.

```
Answer = function() { console.log("constructing...") };
Answer.prototype = { value = 42; };
x = new Answer();
x.value // 42
```

When using constructors, you can use the `instanceof` operator to check if an object was produced by a constructor. `x instanceof c` means `x.__proto__ == c.prototype` .

Finally, for the more curious amongst you, here is an illustration that gives some more details (source (http://www.mollypages.org/misc/js.mp)).



Exercise

- Implement `Object.create` using the `new` operator.
- Implement the functionality of `new` using `Object.create`.

Do use an explicitly created object as prototype (don't use `Object.prototype` for instance). For the first exercise, you should be able to reproduce the following scenario:

```
proto = {id: 42};
x = myCreate(proto);
x.__proto__ == proto // true
x.id // 42
```

Enumerating Objects

You can iterate over the properties of an object using the `for .. in` construct.

```
x = { a: 1, b: 2}
for (var prop in x) { console.log(prop) } // prints 'a' then 'b'
```

You may have noticed that we are not getting any of the properties inherited from `x` 's prototype. This is **not** because the for loop excludes inherited properties. We don't see properties "inherited" from `Object.prototype` because they all have the `enumerate` attribute set to `false`.

Indeed, each object can associate some "attributes" to each of its properties. These attributes control among other things if the properties can be enumerated, written to or deleted, and can be used to associate a getter and a setter with the property. You can learn more about attributes here (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) .

You can use the following functions to list all the properties of an object, even if they are not enumerable. `ownPropos(obj)` will give all properties held by `obj` but not by its prototype chain. It will include properties that hide a similarly-named property in the prototype chain. `allProps(obj)` will list all properties accessible on `obj` , including properties from its prototype chain. This may cause some properties to appear multiple times if `obj` or a prototype has a property hiding a property defined in a prototype defined further down the chain.

```
global.ownProps = Object.getOwnPropertyNames;

global.allProps = function(obj) {
    var p = [];
    while (obj) {
        p.push.apply(p, ownProps(obj));
        obj = Object.getPrototypeOf(obj);
    }
    return p;
};
```

Scoping

There a few pitfall with scoping in Javascript. Javascript recognizes three scopes: global, function, and eval. We won’t talk about eval (because eval is evil). Crucially, there is no block scope!

```
y = 1
if (true) { var y = 2; }
y // 2
```

Here we would have expected the `y` in the if-block to shadow the other `y` . Instead, it refers to the same variable.

It gets worse. Javascript implements a mechanism called “hoisting”. What it does is that it automatically lifts all variable declaration (with `var`) to the top of the innermost function in which they appear, or to the top of the script if they appear at the top-level.

Here’s another take on our last example:

```
y = 1
(function() {
    y = 2;
    if (true) { var y = 3; }
    return y;
})() // 3
y // 1
```

We would have been forgiven to think that we assigned the global `y` in our function, but in fact we’re affecting the local ’y`, whose declaration comes later, and in a block besides.

The top level also have a few pitfalls. Did you notice that you could write `x = {};` in the REPL, even if you haven’t called `var x;` before?

In fact, unqualified identifiers that are not local variables are always a shorthand to properties of a global object. In Node.js, this global object is called `global`. In browsers, it is called `window`. So if `x` appears somewhere where no `var x` is in scope, it really means `global.x`.

There’s more. If you do `var x` at the top level, Javascript will also add a `x` property to the global object. Except this time you cannot delete it with `delete global.x`.

Closures

Javascript support closures, so you can keep reference to local variables inside functions, and even share them amongst multiple functions.

```
(function() {
    var x = 0;
    global.incX = function() { ++x; };
    global.readX = function() { return x; }
})();
incX()
readX(); // 1
```

The Functional, Evented Nature of Javascript

Before getting to the exercises, a word about how Javascript is usually used, and how to use it effectively.

Javascript is event-driven. This means that after executing a script, the Javascript engine enters the “event loop”. There, it waits for event to happens and executes callbacks (Javascript functions) upon reception of events.

In the browser, events are mostly input events: clicks, key presses, … A few other kind of events are possible, like timer expirations. The whole idea behind Node.js is to apply the same kind of event-driven framework to server-side applications. On the server’s side, events typically signal the completion of an I/O operation, such as reading data from a file or from a socket, or obtaining the result of an HTTP request. We will explore this in the exercises.

Since functions play such an important role in the evented model, it pays to be able to use them effectively. Fortunately, the standard API (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects) supplies a few higher-order functions (functions that take other functions as parameter) to support this.

There are many libraries that expand the rather meager offering of the standard API; the most famous of which is Underscore.js (http://underscorejs.org/) . We won’t be using Underscore.js in the exercise – there’s enough to learn as it is.

Node.js Exercises

And now, for the real deal.

Run the command `npm install -g learnyounode` and then `learnyounode`. This will launch a menu that offers you a series of challenges to learn the basics of the Node.js API.

You probably don’t have the time to do all the exercises, so go as far as you can :)

More Things

Here are a few things that were not explained here or in the tutorials but that you may want to learn. Here we limit ourselves to the Javascript *language* itself, but there a gazillion other things (libraries, frameworks) connected to Javascript that may catch your interest.

- Exception handling (`try ... catch` and `throw`).
- Handling variable number of parameters (the `arguments` special variable).
- Strict mode, which prevents you from doing things you probably shouldn’t be doing anyway.

And here are a few learning resources.

- The Mozilla Developer Network JavaScript Reference (https://developer.mozilla.org/en-US/docs/Web/JavaScript) . This includes guides as well as a full reference to the language’s Standard API.
- Javascript: The Best Parts (http://www.amazon.com/JavaScript-Good-Parts-Douglas-Crockford/dp/0596517742) . A short and to the point book on how to use Javascript without selling your mind to Beelzebub.
- Nodeschool (http://nodeschool.io/) . A set of exercises like the one we used to practice on Node.js. We especially recommend the module on functional Javascript.