Photo from: lifeofpix.com

JavaScript

Functions, Closures, and Prototypes

JavaScript Functions, Closures, and Prototypes

Amin Meyghani

This book is for sale at http://leanpub.com/javascript-closure

This version was published on 2016-07-08

Contents

1. Preface

There is no doubt that JavaScript has grown to become one of the most important programming languages today. Yet, surprisingly enough, there are many who consider methodical learning of JavaScript unnecessary. This is very surprising, even so now in 2016 that JavaScript has improved dramatically since 1995 when it was first released. This trend seems to be prevalent in many venues, but it doesn't have to be that way. One of the goals of this book is to point you in the right direction by focusing on the three very important concepts in JavaScript, namely functions, closures, and prototypes. Hopefully, by the end of the book you will feel comfortable with these concepts and find yourself enabled to to go ahead and use them effectively in your own programs. In addition, there is a bonus chapter listing interview questions that are often asked in many interviews for any JavaScript or front-end related position. You can use that chapter to evaluate yourself and revisit the chapters or sections as necessary.

1.1 Book overview

In *Chapter 1* we will explore all the fundamental concepts related to functions, including function objects, function declarations and expressions, the `this` keyword and more.

In *Chapter 2* we will learn what closures are, why they are useful, and how you can use them in your programs.

In *Chapter 3* we will look at probably the most important concept in JavaScript, that is the prototypes. We will learn what they are, why they make JavaScript so different than class-oriented languages, and how you can use them to simplify the design of your programs.

1.2 How to read this book

Every chapter of the book opens with a list of bullet points of important concepts. Each bullet point is then explored in detail through the chapter. You may find the book very dry in nature, so may want to take breaks while reading. Although you may want to read chapters out of order, or just skim through the chapters, please don't. Please take your time, study each chapter slowly, and take breaks frequently if you find yourself unmotivated. The book is designed this way to serve as a reference so that you can refer to any time in the future. Also, keep in mind that the book is always up-to-date with the latest spec of the language, which is currently ES2015. I genuinely hope you get something out of the book, and please submit an issue if you don't understand something, or if you find a spelling error or a mistake in the code. The book is written in Markdown and is hosted in github: https://github.com/aminmeyghani/js-fn-book

Happy coding :)

1.3 Requirements

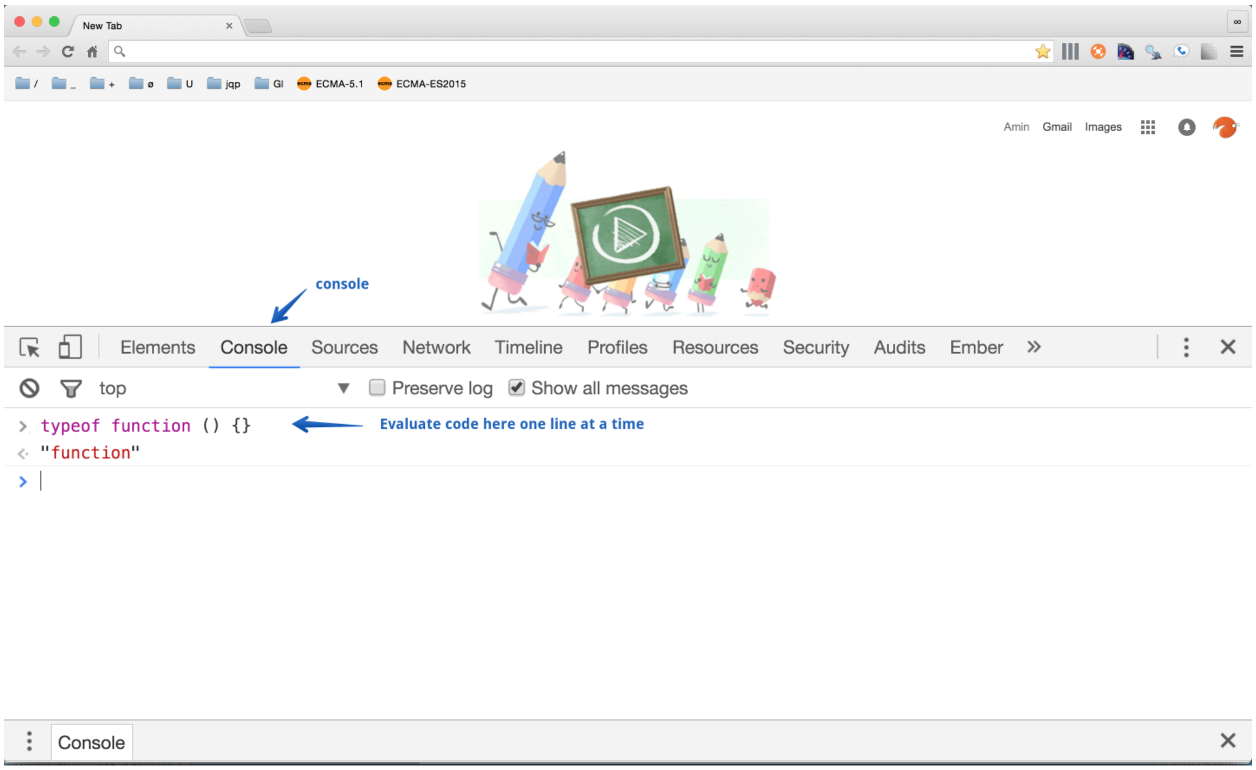
In order to follow along with the book, you need the following:

- Google Chrome
- Node > 5
- A text editor (Sublime Text is recommended, but not necessary)

Below, you can find useful information about Chrome DevTools and installing Node. Since we will be using them throughout the book, please take the time to make sure that you have all of them set up.

Using Chrome DevTools

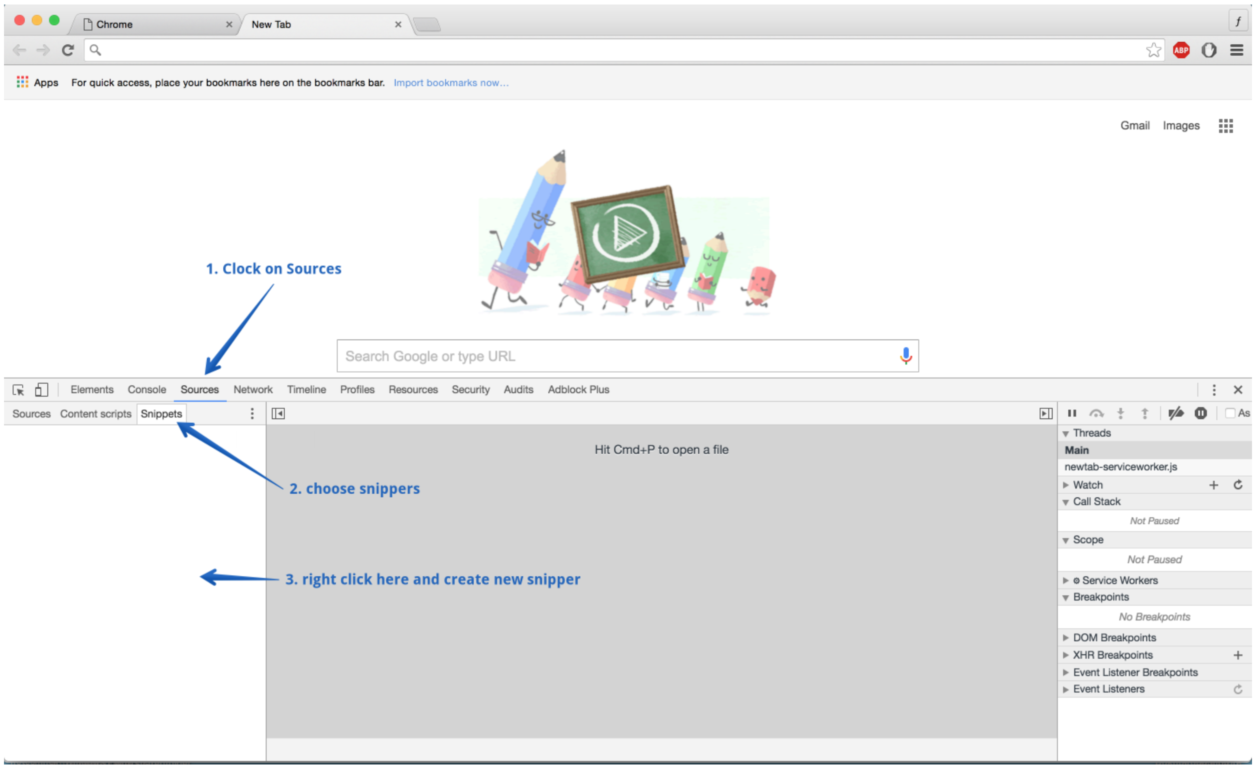
You can run JavaScript code in Chrome DevTools. You can either use the console or the snippets tab to create snippets of JavaScript code. In order to use the console, open Chrome, right click on the page, and choose inspect. Once you do that, the DevTools opens up. Once the DevTools is open, click on the Console tab to open the Console. Once the Console tab opens, you can run JavaScript code one line at a time. The screenshot below demonstrates how to find the Console tab:



```
> typeof function () {}
< "function"
>
```

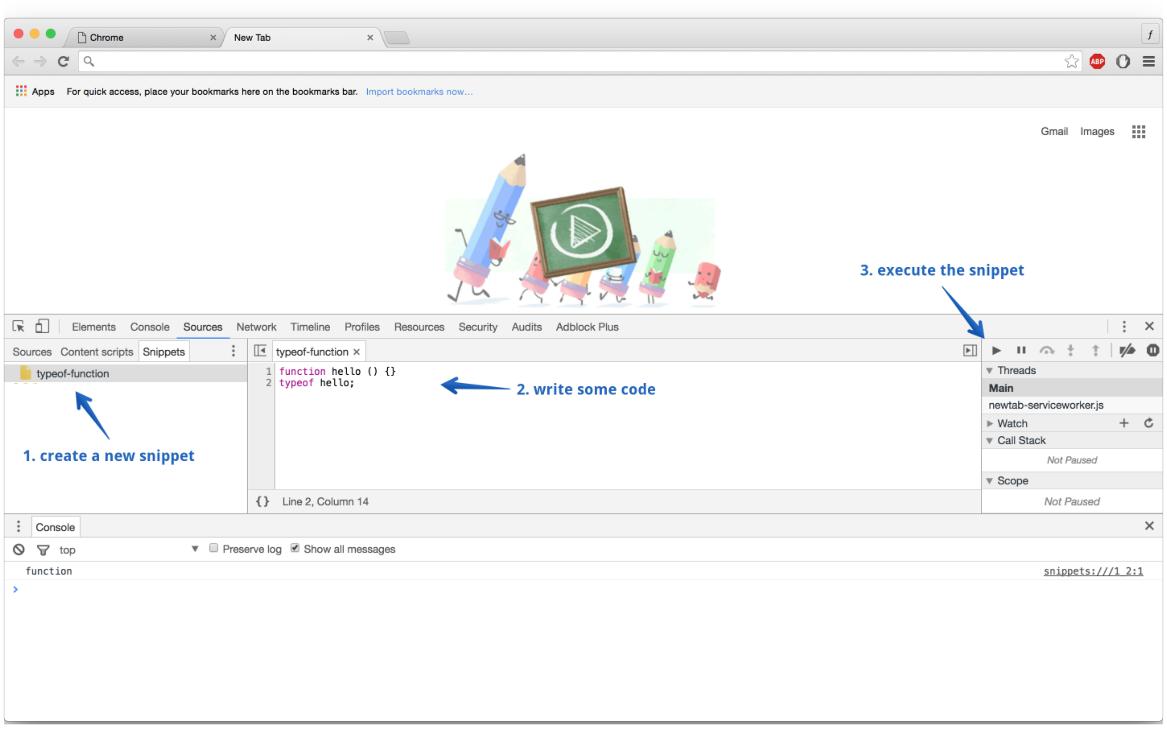
Using the Console

In addition to the Console, you can use the snippets tab to create snippets of code and execute them. In order to create a new snippet, click on the Snippet tab, and then right click in the empty area to create a new snippet. The screenshot below shows you how to create a new snippet:



Creating a new snippet

After you created a snippet, you can write some code and execute it by either clicking on the run button on the right hand side of the DevTools, or using the `command + enter` shortcut. The screenshot belows demonstrates how to run the snippet:



```
function hello () {}
typeof hello;
```

```
function
```

Executing a snippet

Installing Node

Node is a JavaScript run-time that you can use to execute JavaScript scripts. The easiest way to install and manage Node is with nvm[1]. You can use the following to install NVM:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.0/install.sh | b\
ash
```

After that, make a new terminal and make sure that `nvm` is installed with `nvm --version`. If you don't get any output, try adding the following to your `~/.bash_profile` file:

```
export NVM_DIR=~/.nvm
source $(brew --prefix nvm)/nvm.sh
```

After NVM is installed, you can use it to install any Node version that you need. For example, to install the latest version 5, run:

[1] https://github.com/creationix/nvm

```
nvm install 5
```

That’s it! Now, if you want to set this version as your machine default, run the following:

```
nvm alias default 5
```

Now that you have Node installed, you can execute Node scripts. For example, make a file on your desktop called `script.js`:

```
touch ~/Desktop/script.js
```

Then add the following to the file:

```
console.log('hello world');
```

Then execute the script with `node ~/Desktop/script.js` and you should see the output in the terminal. That’s it.

TODO

2. Introduction

Functions, closures, and prototypes are essential in learning JavaScript.

3. Functions

Functions are callable objects. It is very important to note that in JavaScript functions are objects. It might be misleading because when you use the `typeof` operator on a function, you get `function` as the output. This is one of the instances where JavaScript lies to you. The output of `typeof function () {}` should be `object` because functions are objects in JavaScript. That makes functions very powerful, because you can think of them as callable objects. You can use a function as an object, a piece of reusable code, or a function that creates objects.

3.1 Creating Functions

There are two main ways of defining functions:

- function declaration
- function expression

If a statement starts with the `function` keyword, then you have a function declaration:

```
function myFn() {}
```

But if you assign a function to a variable, you have a function expression:

```
const fnRef = function myFn() {};
```

Note that you need a semi colon at the end of a function expression, but there are no semi colons at the end of function declarations.

3.2 Function Inputs and Outputs

You can pass values into and return values from a function. For example, you can make a function that takes two numbers and `returns` the result of adding them:

```
function add(a, b) {
  return a + b;
}
```

If you notice, the inputs to the function are placed inside the parenthesis separated by commas. Note that `a` and `b` represent the inputs to the function. If you don't specify any return values for your function, JavaScript will return `undefined` by default:

```
function fn() {
  const a = 1;
  // other stuff.
  // no return.
}
fn(); // `undefined` is returned.
```

Note that arguments are always passed by value to functions. The value could be a primitive or a reference value in the case of non-primitives. This means that if you assign the argument to another value or thing, it will not affect the thing outside of the function:

```
const n = 1;
function change(x) {
  x = 5;
}
console.log(n); // 1;
change(n);
console.log(n); // 1;
```

It doesn't matter what is passed in, the behavior is consistent, even if the input is a non-primitive (an object):

```
const n = {value: 1};
function change(x) {
  x = {};
}
console.log(n); // {value: 1};
change(n);
console.log(n); // {value: 1};
```

However, if you decide to change the property of the object that the reference is pointing to, the object will be mutated:

```
const n = {value: 1};
function mutate(x) {
  x.value = 22;
}
console.log(n); // {value: 1}
mutate(n);
console.log(n); // {value: 22}
```

As you can see, arguments passed to functions are always copied. In the case of primitives, the actual values are copied, and the case of non-primitives, the reference is copied and becomes available in the function and assigning the argument to another thing does not change the original value.

The `arguments` Object

Every function in JavaScript has access to the magical `arguments` object inside the function body that contains the arguments passed to a function. Let's look at a basic example to demonstrate how you can access the `arguments` object.

In this example, we are going to create a function called `sum` that is going to simply return the `arguments` object when it is called:

```
function sum() {
  return arguments;
}
```

When you call the function with `sum(1,2,3)`, you can see that `arguments` is an object with three keys and values:

```
{
  '0': 0,
  '1': 1,
  '2': 2
}
```

So it is important to note that the `arguments` object is not an array object. This means that `arguments` does not inherit array methods and the following returns false:

```
const args = sum(1,2,3);
Object.getPrototypeOf(args) === Array.prototype; // -> false
```

As of ES2015, you can use the `Array.from` method to convert an iterable object like the `arguments` object to an array. Alternatively, you can call the `slice` method of `Array.prototype` in the context of the `arguments` object to return an array containing the values:

```
const args = sum(1,2,3);
const argArray = Array.prototype.slice.call(args); // -> [1,2,3]
Object.getPrototypeOf(argArray) === Array.prototype // -> true
```

3.3 Executing a Function

A function can be executed. This is generally known as calling a function. You can call a function by using the name of the function, followed by `()`:

```
add();
```

Notice that we are calling the functions without any inputs. Let's give the function to numbers as inputs:

```
add(1,2); // -> 3
```

In addition to the `()` operator, there are 3 other ways to invoke a function. That is, using:

- `call`
- `apply`
- and the `new` keyword before invoking the function with `()`

So for our simple example, let's call the function using all these methods and explore when you would want to use each method.

`Function.prototype.call`

When you use `call` to invoke a function, the first argument is the value of the context object. And the second argument is the parameters separated by commas:

```
add.call({}, 1,2); // -> 3
```

`Function.prototype.apply`

Using `apply` is similar to `call`, except instead of passing the arguments one by one separated by commas, you pass the arguments in an array:

```
add.apply({}, [1,2]); // -> 3
```

Calling Function with `new`

You call a function with the `new` keyword when you want to use the function to create objects. So you wouldn't really call the `sum` function above with the `new` keyword. When a function is called with the `new` keyword, couple of things happen behind the scenes:

- A new empty object is created
- The context object `this` is bound to the new empty object
- The new object is linked to the function's prototype property
- `this` is automatically returned unless another value is returned explicitly from the function

This is the mechanism provided by JavaScript to link objects with prototype objects. We will learn more about them later in the book, but for now just remember that these functions are known as constructor functions and by convention, their name starts with an uppercase letter. Let's look at a simple constructor function that can be used to create `Car` objects:

```
function Car() {
  this.color = 'black';
}
const myCar = new Car();
myCar.color; // -> 'black'
```

As you can see, in the function we use the context object `this` to assign the `color` property. And then we call the function using the `new` keyword and assign the result to the `myCar` variable. Now, `myCar` is the object created by the `Car` constructor function.

Context Object `this`

The context object, `this` is a dynamic object which is set dynamically depending on the way a function is called. There are several rules to follow in order to figure out what `this` is bound to.

First, check if the function of interest is called with the `new` keyword. If so, then `this` is bound to the new object created by the function. If not, check if the function is being called with `call` or `apply`, if so, the first argument tells you what the value of `this` is bound to. If not, check if the function is called in the context of another object, `(someObj.fn())`, if so, then `this` is bound to the object. If none of these cases are met and the function is simply invoked without any context, `this` will be bound to the global object in `non-strict` mode. However, in the `strict mode`, the value of `this` will be `undefined`.

Let's look at some examples and see how the context object is bound to.

Call with `new`

The first example, is just a review from the previous section. That is calling a function with the `new` keyword. When you can a function with the `new` keyword, the context object is bound to the instance object created by the function:

```
function Car() {
  this.color = 'Black';
}
const myCar = new Car();
```

To figure out how `this` is bound, we look at the line where the function is called. In this case, the function is called with the `new` keyword, so then we know that the context is bound to the new object that is going to be returned by the function. This is the case that you always have to check first, because it over rules the other ones.

In the next example, we are going to call a function with `call` and `apply`.

Using `call` or `apply`

Let's see how the value of `this` gets bound when we call a function with `call` or `apply`. For this example, we are going to create a simple function called `printMessage` that is going to read the `name` property on this and return a message:

```
function printMessage(msg) {
  return msg + ' ' + this.name;
}
const message = printMessage.call({name: 'Amin'}, 'Welcome!');
// -> 'Welcome! Amin'
```

In the example above, we look at the line where the function is called and you can see the function is called by `call` and the value of `this` is explicitly passed as the first parameter. So inside the function, `this` will be bound to `{name: 'Amin'}` and when we call the function, it will return the following result:

```
'Welcome! Amin'
```

The `apply` case is very similar to the case above, except we would pass the arguments in an array, but the value of `this` will still be bound to the first argument passed, that is: `{name : 'Amin'}`:

```
function printMessage(msg) {
  return msg + ' ' + this.name;
}
const message = printMessage.apply({name: 'Amin'}, ['Welcome!']);
```

The output is the same as the `call` example, and the important thing to note here is that when you call a function with `call` or `apply`, the first argument is the context object that will be used inside the function as `this`.

Implicit Binding

The next case, to look in order, is when a function is called in the context of another object. So first let's create an object and define a method on it and then call the method in the context of the object:

```
const util = {
  name: 'Utility',
  getName: function () {
    return this.name;
  }
};
const name = util.getName(); // -> Utility
```

Again first we look at the line where the function is called to determine how `this` is bound to. When you look at the area of function call, you can see the function is called in the context of the `util` object, that is `util.getName()`. So the context object is implicitly bound to the `util` object.

The last case, is when a function is called without any context. Using the sample `util` object above, we can assign the `getName` function to a variable and then call the function without any context:

```
const util = {
  name: 'Utility',
  getName: function () {
    return this.name;
  }
};
const getName = util.getName;
const name = getName();
```

In this case, when the function is called, the context object by default is set to the global object. So if you are in the browser, `this` will be bound to the `window` object, and if you are on the server, `this` will be bound to the `global` object.

It is important to note however, in `strict mode` the value of `this` will be undefined if you call a function as is:

```
function setName(name) {
  'use strict';
  this.name = name;
}
setName();
```

When you run the function above, we will get the following error:

```
TypeError: Cannot set property 'name' of undefined
```

The reason is that in `strict mode`, `this` will be set to `undefined` when you call the function without a context. So setting the property on something that is `undefined` is going to throw an error. This is a good thing because then you wouldn't end up setting properties inadvertently on the global object.

3.4 Functions as objects

You can use functions as plain objects:

```
var fnRef = function fnObject () {};
fnRef.someProp = 'foo';
fnRef.hello = function () {
  return 'hello';
};
```

Note that you can then access the properties and methods of this function object:

```
fnRef.someProp// -> 'foo'
fnRef.hello() // -> 'hello'
```

TODO more coming soon …

4. Closures

Generally speaking, a closure is a dynamic reference that is created when an inner function uses variables of an outer function. And to use the closure, the inner function is returned and when closure is called, the variables maintain their previous value state. Let's demonstrate this with a simple example. First, we need an outer function that defines a variable called `stateVal`:

```
function makeClosure() {
  let stateVal = 0;
}
```

After that, we need to create a another function inside that references `stateVal`:

```
function makeClosure() {
  let stateVal = 0;
  function cl() {
    stateVal += 1;
    return stateVal;
  }
}
```

and then, to access the closure function, we need to return it from `makeClosure`:

```
function makeClosure() {
  let stateVal = 0;
  function cl() {
    stateVal += 1;
    return stateVal;
  }
  return cl;
}
```

Now that we have our closure maker function, we can call it and get access to the closure function inside:

```
const inc = makeClosure();
```

Note that `inc` is a closure because it references the `stateVal` variable inside the outer function, so `stateVal` maintains its state every time `inc` is called. So if we call the `inc` function 3 times, we expect the value of `stateVal` to be three at the end:

```javascript
inc();
inc();
const val = inc();
```

At the end of the last `inc` call, the value returned is going to be `3`.

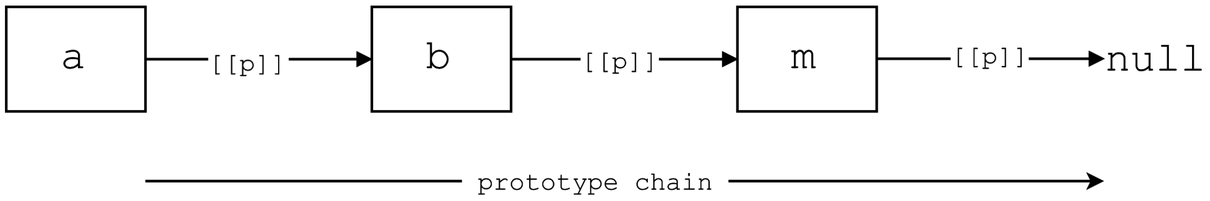
TODO more coming soon ...

5. Prototypes

There are different ways to interpret the prototype concept in JavaScript, but you can explain in it very easily in one sentence:

> “In JavaScript, prototypes are objects that facilitate linking of objects and delegation of methods or properties”

That’s really it. The most confusing part about the prototype concept is just the terminology, but the concept itself is pretty straightforward. The diagram below demonstrates the idea in more detail:



The Prototype Concept

- The boxes represent objects
- The arrow shows the link direction: `a` links to `b`, `b` links to `c`, `c` links to `m`
- The links form the prototype chain for these objects
- `m` is the mother object which contains all the methods common to all objects
- `[[p]]` is the internal property of an object that points to another object. Confusingly enough, this property is known as the `[[prototype]]` in the specification
- In this diagram, `b` is known as the prototype of `a`, `c` the prototype of `b` and `m` the prototype of `c`.
- Property and method look up follows the prototype chain. That is, if you want to look up a property in `a`, JavaScript will first look at `a` itself. If it cannot find it there, it will then look at `b`. If it can’t find it in `b`, it will then look in `c`. If it can’t find it there, it will look in `m`, and eventually if it can’t find it in `m`, it will return `undefined` because `m` is the ‘mother object’ and is not linked to any other object.
- But if you want to look up a property in `c`, JavaScript will not look into `a` or `b`. It will follow the link direction and if it can’t find it in `c`, it will look in `m`, and will return `undefined` if it cannot find it in `m`.
- The `m` object is a special object which comes from `Object.prototype`. But don’t worry about it now, we have enough prototype jargon lying around already :)

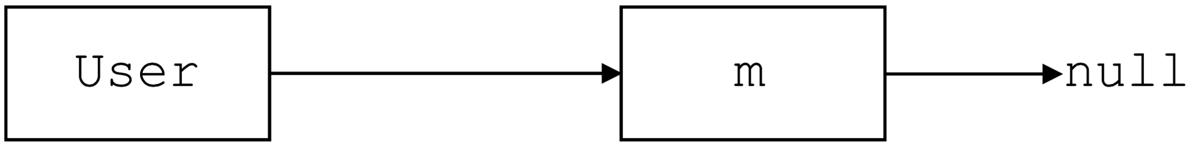
That was the Prototype concept in a nutshell. Now let’s dive in and learn how we can link objects.

5.1 Linking Objects

When you create any objects, behind the scenes JavaScript links your object to the 'mother object' automatically:

```
var user = {
  name: 'Amin'
};
```

Roughly speaking, this is what happens behind the scenes:



Creating a plain old JavaScript Object

- The `user` object gets linked to the 'mother' object
- Object `m` is known as the prototype of `user`
- You can get the prototype of `user` using the `Object.getPrototypeOf` method, that is: `Object.getPrototypeOf(user)`
- The 'mother' object is the `prototype` property of the `Object` constructor function object (WO!)
- You can double check the prototype of `user` using: `Object.getPrototypeOf(user) === Object.prototype // -> true`

As you can see, the terminology can get very confusing, so let's not get caught up with the jargon here. What's important is that fact that objects can be linked to each other and an object can delegate responsibilities to other objects. That is the core of the idea and don't let the jargon confuse you.

Now that you know a little bit about what happens behind the scenes, let's create our own objects and link them together. In JavaScript there are two main ways to link objects. The first one has been available since ES5 and that is `Object.create`. Let's create two objects `a` and `b`, and link `a` to `b`. Because we are linking `a` to `b`, object `b` will be the prototype of `a`:

```
// a -> b: b is the prototype of a.
// defining object `b` who is going to be the prototype of `a`
var b = {
  hello: function () {
    return 'hello';
  }
};
// create `a` and link it to `b`
var a = Object.create(b);
```

That's all it takes to link `a` to `b`. So now if we call `a.hello()`, first JavaScript will look at `a`, and because the `hello` method doesn't exist on `a`, it will then look at `b` and it will invoke the method. Also remember that `b` is automatically linked to the 'mother' object, so when we call, `a.toString()`, JavaScript will look up all the way from `a` to `m` and then it will invoke the method using the implementation in `m`. You can override any method, so let's provide our own `toString` method for `a`:

```
a.toString = function () {
  return 'I am a.';
};
```

So now when we call `a.toString` we get back `I am a`. Now let's look at the other common way to link objects.

The other way of linking objects is using constructor function objects. We are going to do the same example as above, but instead we are going to use a more common way of creating it. So first, let's create the prototype object, that is object `b`. The funny thing is, to create this object first you need to create a function:

```
var F = function () {};
```

Then, you need to get access to a special property on the function, which is created when you create any function in JavaScript. This property, not surprisingly, is called `prototype`:

```
var b = F.prototype;
b.hello = function () {
  return 'hello';
};
```

Now that we have a prototype object, we can create object `a` by invoking the `F` function using the `new` keyword:

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

Because this pattern is so common, you don't need to create a reference to `F.prototype`. You can just add methods to `F.prototype` and then create an object using the function:

```
var F = function () {};
F.prototype.hello = function () {
  return 'hello';
};
var a = new F();
// a -> F.prototype -> Object.prototype -> null
// a ->       b     ->          m       -> null
a.hello(); // -> 'hello'
```

That's it! Now we have linked `a` to `b` and `b` is the prototype of `a`. It might look a little bit weird but that's the mechanism that JavaScript uses to link objects. Maybe now it makes more sense why the `m` object is `Object.prototype`. The reason is that `Object` is a function who, just like any other function, has a property called prototype. So when you create a plain old object for example, it gets linked to `Object.prototype`. It is equivalent to the following:

```
var myObj = Object.create(Object.prototype);
// or
var myObj = new Object();
// Object.getPrototypeOf(myObj) === Object.prototype // -> true
```

The great thing is that this mechanism is consistent throughout JavaScript and that explains a lot of things about JavaScript. Let's explore those in the next section.

5.2 Prototype Objects Inside JavaScript

JavaScript internally has several functions that it uses as constructors. The most common ones are:

- `Boolean`
- `Number`
- `String`
- `Object`
- `Function`

Note that all of these functions, correspond to the fundamental data types in JavaScript:

- Primitive Data Types:
  - boolean
  - number
  - string
- Non-primitive Date Types
  - object (functions are objects too)

The most important point here is that all the methods that are available on the data types come from the value of the `prototype` property of the constructor function. Also note that constructor functions always start with an uppercase letter. Let's look at some examples.

You know that strings are primitive values, yet you can call methods on strings:

```
var str = 'hello';
str.replace('o', 'oo'); // -> helloo
```

First of all, how can we call the `replace` method on the string, if `str` is a primitive value? Second of all, where does the `replace` method come from? In order to answer the first question, we need to learn how JavaScript can coerce one type to another. When you create a primitive string value and assign it to the variable `str`, nothing happens until you use the dot operator on the reference. Then JavaScript coerces the primitive string to the equivalent object by wrapping it with the corresponding constructor function. So in this case, roughly speaking, JavaScript does the following to the primitive value:

```
var str = new String('hello');
```

And that's why you can call the method on `str` because it is an object for a short period of time until it is garbage collected. Now, that we know what happens behind the scenes, it might be easier to answer the second question: where does the `replace` method come from ? The answer is, it comes from `String.prototype` and if you think about it, the answer is consistent to what we learned in the previous section. And since JavaScript is dynamic, you can add a method to `String.prototype` object and magically all the pre existing strings in your code, have a new method. This is not generally recommended, but let's demonstrate this by adding a new method on `String.prototype` called `first` that returns the first character of a given string:

```
String.prototype.first = function () {
  return this.charAt(0);
};
str.first(); //-> 'h'
```

As you can see, this mechanism is consistent across JavaScript, that is, for anything out there in JavaScript, there is a constructor function that has a prototype property that contains the methods to be used by all the instances. If you understand this concept, you will understand the core of JavaScript and how it works. Now, just for fun, here are some more objects with their corresponding prototype chains:

- `Array`

```
const nums = [1,2,3];
// nums -> Array.prototype -> Object.prototype -> null
```

You can get the list of properties on `Array.prototype` with:

```
Object.getOwnPropertyNames(Array.prototype);
//->
/*
["length", "constructor", "toString", "toLocaleString", "join", "pop", "push", "\
reverse",
"shift", "unshift", "slice", "splice", "sort", "filter", "forEach", "some", "eve\
ry",
"map", "indexOf", "lastIndexOf", "reduce", "reduceRight", "copyWithin", "find",
"findIndex", "fill", "includes", "entries", "keys", "concat", "values"]
 */
```

It is interesting to note here that Arrays have their own `toString` method that is different from `Object.prototype.toString` method.

- `Promise`

```
const myPromise = new Promise((resolve, reject) => {
  resolve('hello')
});
// myPromise -> Promise.prototype -> Object.prototype -> null
```

- `Date`

```
const today = new Date();
today.getTime(); // -> 1467023558413
// today -> Date.prototype -> Object.prototype -> null
```

Now that we have a better understanding of the prototype concept in JavaScript, let's dive deeper and explore more interesting topics.

5.3 Inheritance

You can mimic the classical inheritance model in JavaScript with prototype objects. We are just going to demonstrate that here and we won't dive into it since the classical inheritance model does not really fit JavaScript but we are just going to show that here because many seem to use this pattern. After that we will demonstrate functional mixins which are much flexible and dynamic way of achieving inheritance in JavaScript.

For this example, we are going to have a `Person` base type, and create a `Worker`, `Designer`, and `Developer` type which all would inherit from `Person`. First, let's draw a simple diagram to understand the relationship between these types:

```
var developer = new Developer();
/*
 * null
 *  □
 * Object.prototype
 *  □
 * Person.prototype
 *  □
 * Worker.prototype
 *  □
 * Developer.prototype
 *  □
 * developer
 */
```

A Developer has all the methods of:

- a developer
- a worker
- a person

Note that all we need to do is to link some objects and these objects are the `prototype` property of each constructor function. That's what makes delegation possible and efficient in terms of memory usage:

```
function Person () {}
Person.prototype.walk = function () {
  return 'walking ...';
}

function Worker() {}
Worker.prototype = Object.create(Person.prototype);

Worker.prototype.work = function () {
  return 'working ....';
}

function Developer () {}
Developer.prototype = Object.create(Worker.prototype);
Developer.prototype.code = function () {
  return 'coding ....';
}

var dev = new Developer();
dev.code(); // 'coding ...'
dev.work(); // 'working ...'
dev.walk(); // 'walking ...'
```

Also note that you can just simply link objects together, and it would have the same effect:

```
var person = {
  walk() { return 'walking...';}
};

var worker = Object.create(person);
worker.work = function () {
  return 'working';
}

var dev = Object.create(worker);
dev.code = function () {
  return 'coding...';
}
dev.code(); // 'coding ...'
dev.work(); // 'working ...'
dev.walk(); // 'walking ...'
```

The difference is that we only have one instance of the developer and we don't have a function to create more instances of a `Developer` for us. You could wrap this in a function, but JavaScript already has a mechanism and that is linking prototype objects and using constructor functions.

Another important thing to remember is using the `new` keyword. Almost in call cases, you want to hide that away and not force the consumers of your API to use `new` to create instances:

```
function Developer (name) {
  if (!(this instanceof Developer)) {
    return new Developer(name);
  }
  this.name = name;
}
```

By doing those, consumers won't have to call the constructor with `new` which is cleaner and also eliminates the possibility of `this` to be bound to the global object. The other important thing to note is how `this` is bound when the function is invoked with the `new` keyword. When the constructor function is invoke with the `new` keyword, the context object, `this` always refers to the instance. In the example above, we assign the name property of the instance to what is passed. Each instance will get their own names but they will all share the same prototype that enables them efficiently call methods without the need of copying methods to each instance.

That's pretty much all you need to know to be able to use prototype objects. However, you should be aware of great libraries out there, such as Stampit[1] that are much more powerful and flexible than the classical approach and allow you to do much more. In the next section we will explore the effect of inheritance through functional mixins.

5.4 Functional Mixins Over Classical Inheritance

The core idea behind mixins is that you can augment an existing entity with another. Now this entity can be anything, a plain object, a function, or a prototype object. You can either mixin objects with each other or mixin an object with functions. In this section we are going to focus on functional mixins since they are very flexible and convenient when working with JavaScript. In addition, functional mixins are very different than classical inheritance and introduce a different a way of thinking.

The idea behind functional mixins is very simple: you define a bunch of functionalities, and you specify the context to which these functions apply. Let's look at a very simple example:

[1]https://github.com/stampit-org/stampit

```
/*
 * Define functionalities
 * Grouped by `fns` here.
 */
function fns() {
  this.getName = function() {
    return this.name;
  };
}

/* Define Type */
function Person(name) {
  if (!(this instanceof Person)) {
    return new Person(name);
  }
  this.name = name;
}

/* Apply the functionalities to the
prototype object of Person */
fns.call(Person.prototype);

/* make an instance of Person */
const person = Person('Amin');

/* call methods */
person.getName() //-> 'Amin'
```

In the example above, we first create a function that contains the functionalities of a type. Then we define the type and finally we apply the functionalities to the type by calling the `fns` function in the context of the type's prototype object. That's all it takes to add functionalities to the type. Now the interesting thing to note here is that you can create more functions and group other pieces of functionalities and then apply it to the type. In order to demonstrate that, let's redo the example in the previous section with this functional mixins.

If you remember from the previous section, we created the `Developer` type that had the functionalities of a `Person`, `Worker`, and `Developer`. So, let's define the functionalities for each of these first:

```javascript
/* Define Person's functionalities */
function personFns() {
  this.walk = function () {
    return 'Walking ...';
  };
  this.getName = function () {
    return this.name;
  };
}

/* Define Worker's functionalities */
function workerFns() {
  this.work = function () {
    return 'Working ...';
  };
}

/* Define Developer's functionalities */
function developerFns() {
  this.code = function () {
    return 'Coding ...';
  };
}
```

Now that we have defined the groups of functionalities, then, we need to define the `Developer` type:

```javascript
/* Define the Developer type */
function Developer(name) {
  if (!(this instanceof Developer)) {
    return new Developer(name);
  }
  this.name = name;
  this.toString = function () {
    return this.name;
  };
}
```

After that, we need to apply each functionalities to the Developer's prototype:

```
/* apply each functionalities to
Developer's prototype */
[personFns, workerFns, developerFns].forEach(fn => {
  fn.call(Developer.prototype);
});
```

Now we can create an instance of the `Developer` and call any of the methods:

```
/* create an instance and call methods */
const dev = Developer('Amin');
console.log(dev.getName());
console.log(dev.walk());
console.log(dev.work());
console.log(dev.code());
console.log('Dev is: ' + dev);
```

After running the code above you should get the following output:

```
Amin
Walking ...
Working ...
Coding ...
Dev is: Amin
```

And that’s how you can use functional mixins to augment a type. Also note that the context object `this`, always refers to the created instance object. Now that we have learned what functional mixins are, let’s see how we can improve the performance by forming a closure around the mixins to cache the result of the first definition call:

```
const personFns = (function() {
  function walk() {return 'Walking...';}
  function getName() {return this.name;}
  return function() {
    this.walk = walk;
    this.getName = getName;
    return this;
  };
}());

const workerFns = (function() {
  function work() {return 'working...';}
  return function() {
    this.work = work;
    return this;
  };
}());

const developerFns = (function() {
  function code() {return 'coding...';}
  return function() {
    this.code = code;
    return this;
  };
}());
```

All we did here was wrapping each function with a closure, the rest of the code is the same. Now, the effect that this has on performance is noticeable because when the outer function is called, there result is always cached because the inner function references the outer function methods.

6. Interview Questions