

Understanding JavaScript OOP

Published 09 October 2011

WARNING

Some parts of this article are made obsolete or not-as-relevant with the publication of ES2015, and these parts need to be revised. All performance commentaries in the article should be ignored.

JavaScript is an object oriented (OO) language, with its roots in the [Self](#) programming language, although it's (sadly) designed to look like Java. This makes the language's really powerful and sweet features stay covered by some pretty ugly and counter-intuitive *work-arounds*.

One such affected feature is the implementation of prototypical inheritance. The concepts are simple yet flexible and powerful. It makes inheritance and behaviourism first-class citizens, just like functions are first-class in functional-ish languages (JavaScript included).

Fortunately, [ECMAScript 5](#) has gotten plenty of things to move the language in the right way, and it's on those sweet features that this article will expand. I'll also cover the drawbacks of JavaScript's design, and do a little comparison with the classical model here and there, where those would highlight the advantages or disadvantages of the language's implementation of prototypical OO.

It's important to note, though, that this article assumes you have knowledge over other basic JavaScript functionality, like functions (including the concepts of closures and first-class functions), primitive values, operators and such.

1. Objects

Everything you can manipulate in JavaScript is an object. This includes `Strings`, `Arrays`, `Numbers`, `Functions`, and, obviously, the so-called `Object` – there are primitives, but they're converted to an object when you need to operate upon them. An object in the language is simply a collection of key/value pairs (and some internal magic sometimes).

There are no concepts of classes anywhere, though. That is, an object with properties `name: Linda, age: 21` is not an instance of the `Object` class (or any class). Both `Object` and `Linda` are instances of themselves. They define their own behaviour, directly. There are no layers of meta-data (i.e.: classes) to dictate what given object must look like.

You might ask: "how?"; more so if you come from a highly classically Object Orientated language (like Java or C#). "Wouldn't having each object defining their own behaviour, instead of a common *class* mean that if I have 100 objects, I will have 100 different methods? Also, isn't it dangerous? How would one know if an object is really an Array, for example?"

Well, to answer all those questions, we'll first need to unlearn everything about the classical OO approach and start from the ground up. But, trust

me, it's worth it.

The prototypical OO model brings in some new ways of solving old problems, in an more dynamic and expressive way. It also presents new and more powerful models for extensibility and code-reuse, which is what most people are interested about when they talk about Object Orientation. It does not, however, give you contracts. Thus, there are no static guarantees that an object `x` will always have a given set of properties, but to understand the trade-offs here, we'll need to know what we're talking about first.

1.1. What are objects?

As mentioned previously, objects are simple pairs of unique keys that correspond to a value – we'll call this pair a `property`. So, suppose you'd want to describe a few aspects of an old friend (say `Mikhail`), like age, name and gender:

(mikhail)

Property	Value
name	'Mikhail'
age	19
gender	'Male'

Objects are created in JavaScript using the `Object.create` function. It takes a parent and an optional set of property descriptors and makes a brand new instance. We'll not worry much about the parameters now.

An empty object is an object with no parent, and no properties. The syntax to create such object in JavaScript is the following:

```
1 var mikhail = Object.create(null)
```

1.2. Creating properties

So, now we have an object, but no properties – we've got to fix that if we want to describe `Mikhail`.

Properties in JavaScript are dynamic. That means that they can be created or removed at any time. Properties are also unique, in the sense that a property key inside an object correspond to exactly one value.

Creating new properties is done through the `Object.defineProperty` function, which takes a reference to an object, the name of the property to create and a descriptor that defines the semantics of the property.

[illegible]

```
11 Object.defineProperty(mikhail, 'gender', { value: 'Male'
12                                     , writable: true
13                                     , configurable: true
14                                     , enumerable: true })
```

`Object.defineProperty` will create a new property if a property with the given key does not exist in the object, otherwise it'll update the semantics and value of the existing property.

BY THE WAY

You can also use the `Object.defineProperties` when you need to add more than one property to an object:

```
Object.defineProperties(mikhail, { name: { value:
'Mikhail'
1                                     , writable: true
2                                     , configurable: true
3                                     , enumerable: true
4 }
5                                     , age: { value: 19
6                                     , writable: true
7                                     , configurable: true
8                                     , enumerable: true
9 }
10 }
11                                     , gender: { value:
12 'Male'
13                                     , writable: true
14                                     , configurable: true
                                     , enumerable: true
                                     }
    })
```

Obviously, both calls are overtly verbose – albeit also quite configurable –, thus not really meant for end-user code. It's better to

create an abstraction layer on top of them.

1.3. Descriptors

The little objects that carry the semantics of a property are called descriptors (we used them in the previous `Object.defineProperty` calls). Descriptors can be one of two types - data descriptors or accessor descriptors.

Both types of descriptor contain flags, which define how a property is treated in the language. If a flag is not set, it's assumed to be `false` – unfortunately this is usually not a good default value for them, which adds to the verbosity of these descriptors.

WRITABLE

Whether the concrete value of the property may be changed. Only applies to data descriptors.

CONFIGURABLE

Whether the type of descriptor may be changed, or if the property can be removed.

ENUMERABLE

Whether the property is listed in a loop through the properties of the object.

Data descriptors are those that hold concrete values, and therefore have an additional `value` parameter, describing the concrete data bound to the property:

VALUE

The value of a property.

Accessor descriptors, on the other hand, proxy access to the concrete value through getter and setter functions. When not set, they'll default to `undefined`.

GET ()

A function called with no arguments when the property value is requested.

SET (NEW_VALUE)

A function called with the new value for the property when the user tries to modify the value of the property.

1.4. Ditching the verbosity

Luckily, property descriptors are not the only way of working with properties in JavaScript, they can also be handled in a simple and concise way.

JavaScript also understands references to a property using what we call *bracket notation*. The general rule is:

```
1<bracket-access> ::= <identifier> "[" <expression> "]"
```

Where `identifier` is the variable that holds the object containing the properties we want to access, and `expression` is any valid JavaScript expression that defines the name of the property. There are no constraints in which name a property can have¹, everything is fair game.

Thus, we could just as well rewrite our previous example as:

```
1mikhail[ 'name' ]    = 'Mikhail'
2mikhail[ 'age' ]     = 19
3mikhail[ 'gender' ]  = 'Male'
```

> **Note** > All property names are ultimately converted to a String, such that ``object[1]``, ``object[[1]]``, ``object['1']`` and ``object[variable]`` (when the variable resolves to ``1``) are all equivalent. `{: .note}`

There is another way of referring to a property called *dot notation*, which usually looks less cluttered and is easier to read than the bracket alternative. However, it only works when the property name is a [valid JavaScript IdentifierName²](#), and doesn't allow for arbitrary expressions (so, variables here are a no-go).

The rule for *dot notation* is:

```
1<dot-access> ::= <identifier> "." <identifier-name>
```

This would give us an even sweeter way of defining properties:

```
1mikhail.name    = 'Mikhail'
2mikhail.age     = 19
3mikhail.gender  = 'Male'
```

Both of these syntaxes are equivalent to creating a data property, with all semantic flags set to `true`.

1.5. Accessing properties

Retrieving the values stored in a given property is as easy as creating new ones, and the syntax is mostly similar as well – the only difference being there isn't an assignment.

So, if we want to check on Mikhail's age:

```
1mikhail[ 'age' ]  
2// => 19
```

Trying to access a property that does not exist in the object simply returns `undefined` ³:

```
1mikhail[ 'address' ]  
2// => undefined
```

1.6. Removing properties

To remove entire properties from an object, JavaScript provides the `delete` operator. So, if you wanted to remove the `gender` property from the `mikhail` object:

```
1delete mikhail[ 'gender' ]  
2// => true  
3  
4mikhail[ 'gender' ]  
5// => undefined
```

The `delete` operator returns `true` if the property was removed, `false` otherwise. I won't delve into details of the workings of this operator, since [@kangax](#) has already written a [most awesome article on how delete works](#).

1.7. Getters and setters

Getters and setters are usually used in classical object oriented languages to provide encapsulation. They are not much needed in JavaScript, though, given how dynamic the language is – ~~and my bias against the feature.~~

At any rate, they allow you to proxy the requests for reading a property value or setting it, and decide how to handle each situation. So, suppose we had separate slots for our object's first and last name, but wanted a simple interface for reading and setting it.

First, let's set the first and last names of our friend, as concrete data properties:

```
1Object.defineProperty(mikhail, 'first_name', { value: 'Mikhail'
2                                     , writable: true })
3
4Object.defineProperty(mikhail, 'last_name', { value: 'Weiß'
5                                     , writable: true })
```

Then we can define a common way of accessing and setting both of those values at the same time – let's call it `name`:

```
1// () → String
2// Returns the full name of object.
3function get_full_name() {
4    return this.first_name + ' ' + this.last_name
5}
6
7// (new_name:String) → undefined
8// Sets the name components of the object, from a full name.
9function set_full_name(new_name) { var names
10    names = new_name.trim().split(/\s+/)
```

```
11     this.first_name = names['0'] || ''
12     this.last_name  = names['1'] || ''
13}
14
15Object.defineProperty(mikhail, 'name', { get: get_full_name
16                                     , set: set_full_name
17                                     , configurable: true
18                                     , enumerable:   true })
```

Now, every-time we try to access the value of Mikhail's `name` property, it'll execute the `get_full_name` getter.

```
1mikhail.name
2// => 'Mikhail Weiß'
3
4mikhail.first_name
5// => 'Mikhail'
6
7mikhail.last_name
8// => 'Weiß'
9
10mikhail.last_name = 'White'
11mikhail.name
12// => 'Mikhail White'
```

We can also set the name of the object, by assigning a value to the property, this will then execute `set_full_name` to do the dirty work.

```
1mikhail.name = 'Michael White'
2
3mikhail.name
4// => 'Michael White'
5
6mikhail.first_name
7// => 'Michael'
8
```

```
9mikhail.last_name  
10// => 'White'
```

Of course, getters and setters make property access and modification **fairly slower**. They do have some use-cases, but while browsers don't optimise them better, methods seem to be the way to go.

Also, it should be noted that while getters and setters are usually used for encapsulation in other languages, in ECMAScript 5 you still can't have such if you need the information to be stored in the object itself. All properties in an object are public.

1.8. Listing properties

Since properties are dynamic, JavaScript provides a way of checking out which properties an object defines. There are two ways of listing the properties of an object, depending on what kind of properties one is interested into.

The first one is done through a call to `Object.getOwnPropertyNames`, which returns an `Array` containing the names of **all** properties set directly in the object – we call these kind of property **own**, by the way.

If we check now what we know about Mikhail:

```
1Object.getOwnPropertyNames(mikhail)  
2// => [ 'name', 'age', 'gender', 'first_name', 'last_name' ]
```

The second way is using `Object.keys`, which returns all own properties that have been marked as **enumerable** when they were defined:

```
1Object.keys(mikhail)
2// => [ 'name', 'age', 'gender' ]
```

1.9. Object literals

An even easier way of defining objects is to use the object literal (also called *object initialiser*) syntax that JavaScript provides. An object literal denotes a fresh object, that has its parent as the `Object.prototype` object. We'll talk more about parents when we visit inheritance, later on.

At any rate, the object literal syntax allows you to define simple objects and initialise it with properties at the same time. So, we could rewrite our Mikhail object to the following:

```
1var mikhail = { first_name: 'Mikhail'
2                , last_name: 'Weiß'
3                , age:      19
4                , gender:   'Male'
5
6                // () → String
7                // Returns the full name of object.
8                , get name() {
9                    return this.first_name + ' ' + this.last_name
10               }
11
12               // (new_name:String) → undefined
13               // Sets the name components of the object,
14               // from a full name.
15               , set name(new_name) { var names
16                                       names = new_name.trim().split(/\s+/)
17                                       this.first_name = names['0'] || ''
18                                       this.last_name  = names['1'] || '' }
19           }
```

Property names that are not valid identifiers must be quoted. Also note

that the getter/setter notation for object literals strictly defines a new anonymous function. If you want to assign a previously declared function to a getter/setter, you need to use the `Object.defineProperty` function.

The rules for object literal can be described as the following:

```
1 <object-literal> ::= "{" <property-list> "}"
2                      ;
3 <property-list>  ::= <property> ["," <property>]*
4                      ;
5 <property>       ::= <data-property>
6                      | <getter-property>
7                      | <setter-property>
8                      ;
9 <data-property>  ::= <property-name> ":" <expression>
10                  ;
11 <getter-property> ::= "get" <identifier>
12                  :   <function-parameters>
13                  :   <function-block>
14                  ;
15 <setter-property> ::= "set" <identifier>
16                  :   <function-parameters>
17                  :   <function-block>
18                  ;
19 <property-name>  ::= <identifier>
20                  | <quoted-identifier>
21                  ;
```

Object literals can only appear inside expressions in JavaScript. Since the syntax is ambiguous to block statements in the language, new-comers usually confound the two:

```
1 // This is a block statement, with a label:
2 { foo: 'bar' }
3 // => 'bar'
```

```
4
5// This is a syntax error (labels can't be quoted):
6{ "foo": 'bar' }
7// => SyntaxError: Invalid label
8
9// This is an object literal (note the parenthesis to force
10// parsing the contents as an expression):
11({ "foo": 'bar' })
12// => { foo: 'bar' }
13
14// Where the parser is already expecting expressions,
15// object literals don't need to be forced. E.g.:
16var x = { foo: 'bar' }
17fn({foo: 'bar'})
18return { foo: 'bar' }
191, { foo: 'bar' }
20( ... )
```

2. Methods

Up until now, the Mikhail object only defined slots of concrete data – with the exception of the name getter/setter. Defining actions that may be performed on a certain object in JavaScript is just as simple.

This is because JavaScript does not differentiate how you can manipulate a `Function`, a `Number` or an `Object`. Everything is treated the same way (i.e.: functions in JavaScript are first-class).

As such, to define an action for a given object, you just assign a function object reference to a property. Let's say we wanted a way for Mikhail to greet someone:

```
1// (person:String) → String
2// Greets a random person
3mikhail.greet = function(person) {
```



```
4     return this.name + ': Why, hello there, ' + person + '.'  
5 }
```

After setting the property, we can use it the same way we used the concrete data that were assigned to the object. That is, accessing the property will return a reference to the function object stored there, so we can just call.

```
1mikhail.greet('you')  
2// => 'Michael White: Why, hello there, you.'  
3  
4mikhail.greet('Kristin')  
5// => 'Michael White: Why, hello there, Kristin.'
```

2.1. Dynamic `this`

One thing that you must have noticed both in the `greet` function, and the functions we've used for the `name`'s getter/setter, is that they use a magical variable called `this`.

It holds a reference to the object that the function is being applied to. This doesn't necessarily mean that `this` will equal the object where the function is **stored**. No, JavaScript is not so selfish.

Functions are generics. That is, in JavaScript, what `this` refers to is decided dynamically, at the time the function is called, and depending only on how such a function is called.

Having `this` dynamically resolved is an incredible powerful mechanism for the dynamism of JavaScript's object orientation and lack of strictly enforced structures (i.e.: classes), this means one can apply a function to any object that meets the requirements of the actions it performs,

regardless of how the object has been constructed – hack in some custom multiple dispatcher and you have CLOS.

2.2. How `this` is resolved

There are four different ways of resolving the `this` variable in a function, depending on how a function is called: directly; as a method; explicitly applied; as a constructor. We'll dive in the first three for now, and come back at constructors later on.

For the following examples, we'll take these definitions into account:

```
1// (other:Number[, yet_another:Number]) → Number
2// Returns the sum of the object's value with the given Number
3function add(other, yet_another) {
4    return this.value + other + (yet_another || 0)
5}
6
7var one = { value: 1, add: add }
8var two = { value: 2, add: add }
```

2.2.1. CALLED AS A METHOD

If a function is called as an object's method, then `this` inside the function will refer to the object. That is, when we explicitly state that an object is carrying an action, then that object will be our `this` inside the function.

This is what happened when we called `mikhail.greet()`. The property access at the time of the call tells JavaScript that we want to apply whatever actions the `greet` function defines to the `mikhail` object.

```
1one.add(two.value) // this === one
2// => 3
3
4two.add(3)          // this === two
5// => 5
6
7one['add'](two.value) // brackets are cool too
8// => 3
```

2.2.2. CALLED DIRECTLY

When a function is called directly, `this` will be resolved to the global object in the engine (e.g.: `window` in browsers, `global` in Node.js)

```
1add(two.value) // this === global
2// => NaN
3
4// The global object still has no `value` property, let's fix that.
5value = 2
6add(two.value) // this === global
7// => 4
```

2.2.3. EXPLICITLY APPLIED

Finally, a function may be explicitly applied to any object, regardless of whether the object has the function stored as a property or not. These applications are done through either the `call` or `apply` method of a function object.

The difference between these two methods is the way they take in the parameters that will be passed to the function, and the performance – `apply` being up to 55x slower than a direct call, whereas `call` is usually not as bad. This might vary greatly depending on the engine though, so it's always better to do a [Perf test](#) rather than being scared of

using the functionality – don't optimise early!

Anyways, `call` expects the object that the function will be applied to as the first parameter, and the parameters to apply to the function as positional arguments:

```
1 add.call(two, 2, 2)           // this === two
2 // => 6
3
4 add.call(window, 4)          // this === global
5 // => 6
6
7 add.call(one, one.value)      // this === one
8 // => 2
```

On the other hand, `apply` lets you pass an array of parameters as the second parameter of the function. The array will be passed as positional arguments to the target function:

```
1 add.apply(two, [2, 2])        // equivalent to two.add(2, 2)
2 // => 6
3
4 add.apply(window, [4])        // equivalent to add(4)
5 // => 6
6
7 add.apply(one, [one.value])    // equivalent to one.add(one.value)
8 // => 2
```

NOTE

What `this` resolves to when applying a function to `null` or `undefined` depends on the semantics used by the engine. Usually, it would be the same as explicitly applying the function to the global object. But if the engine is running on [strict mode](#), then `this` will be

resolved as expected – to the exact thing it was applied to:

```
1window.value = 2
2add.call(undefined, 1) // this === window
3// => 3
4
5void function() {
6  "use strict"
7  add.call(undefined, 1) // this === undefined
8  // => NaN
9  // Since primitives can't hold properties.
10}()
```

2.3. Bound methods

Aside from the dynamic nature of functions in JavaScript, there is also a way of making a function bound to an specific object, such that `this` inside that function will always resolve to the given object, regardless of whether it's called as that object's method or directly.

The function that provides such functionality is `bind`. It takes an object, and additional parameters (in the same manner as `call`), and returns a new function that will apply those parameters to the original function when called:

```
1var one_add = add.bind(one)
2
3one_add(2) // this === one
4// => 3
5
6two.one_adder = one_add
7two.one_adder(2) // this === one
8// => 3
9
10one_add.call(two, 2) // this === one
```

3. Inheritance

Up to this point we have seen how objects can define their own behaviours, and how we can reuse (by explicit application) actions in other objects, however, this still doesn't give us a nice way for code reuse and extensibility.

That's where inheritance comes into play. Inheritance allows for a greater separation of concerns, where objects define specialised behaviours by building upon the behaviours of other objects.

The prototypical model goes further than that, though, and allows for selective extensibility, behaviour sharing and other interesting patterns we'll explore in a bit. Sad thing is: the specific model of prototypical OO implemented by JavaScript is a bit limited, so circumventing these limitations to accommodate these patterns will bring in a bit of overhead sometimes.

3.1. Prototypes

Inheritance in JavaScript revolves around cloning the behaviours of an object and extending it with specialised behaviours. The object that has it's behaviours cloned is called **Prototype** (not to be confounded with the `prototype` property of functions).

A prototype is just a plain object, that happens to share it's behaviours with another object – it acts as the object's parent.

Now, the concepts of this *behaviour cloning* does not imply that you'll

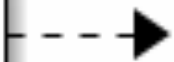
have two different copies of the same function, or data. In fact, JavaScript implements inheritance by delegation, all properties are kept in the parent, and access to them is just extended for the child.

As mentioned previously, the parent (or `[[Prototype]]`) of an object is defined by making a call to `Object.create`, and passing a reference of the object to use as parent in the first parameter.


This would come well in our example up until now. For example, the greeting and name actions can be well defined in a separate object and shared with other objects that need them.

Which takes us to the following model:

(person)

Property	Value
[[Prototype]]	 nil
name	[Getter/Setter]
greet	[Function]

(mikhail)

Property	Value
[[Prototype]]	
first_name	'Mikhail'
last_name	'Weiß'
age	19
gender	'Male'

We can implement this in JavaScript with the following definitions:

```
1 var person = Object.create(null)
2
3 // Here we are reusing the previous getter/setter functions
4 Object.defineProperty(person, 'name', { get: get_full_name
5                                     , set: set_full_name
6                                     , configurable: true
7                                     , enumerable: true })
8
9 // And adding the `greet` function
10 person.greet = function (person) {
11     return this.name + ': Why, hello there, ' + person + '.'
```

```
12}
13
14// Then we can share those behaviours with Mikhail
15// By creating a new object that has it's [[Prototype]] property
16// pointing to `person`.
17var mikhail = Object.create(person)
18mikhail.first_name = 'Mikhail'
19mikhail.last_name  = 'Weiß'
20mikhail.age        = 19
21mikhail.gender     = 'Male'
22
23// And we can test whether things are actually working.
24// First, `name` should be looked on `person`
25mikhail.name
26// => 'Mikhail Weiß'
27
28// Setting `name` should trigger the setter
29mikhail.name = 'Michael White'
30
31// Such that `first_name` and `last_name` now reflect the
32// previously name setting.
33mikhail.first_name
34// => 'Michael'
35mikhail.last_name
36// => 'White'
37
38// `greet` is also inherited from `person`.
39mikhail.greet('you')
40// => 'Michael White: Why, hello there, you.'
41
42// And just to be sure, we can check which properties actually
43// belong to `mikhail`
44Object.keys(mikhail)
45// => [ 'first_name', 'last_name', 'age', 'gender' ]
```

3.2. How `[[Prototype]]` works

As you could see from the previous example, none of the properties defined in `Person` have flown to the `Mikhail` object, and yet we could

access them just fine. This happens because JavaScript implements delegated property access, that is, a property is searched through all parents of an object.

This parent chain is defined by a hidden slot in every object, called `[[Prototype]]`. You can't change this slot directly⁴, so the only way of setting it is when you're creating a fresh object.

When a property is requested from the object, the engine first tries to retrieve the property from the target object. If the property isn't there, the search continue through the immediate parent of that object, and the parent of that parent, and so on.

This means that we can change the behaviours of a prototype at run time, and have it reflected in all objects that inherit from it. For example, let's suppose we wanted a different default greeting:

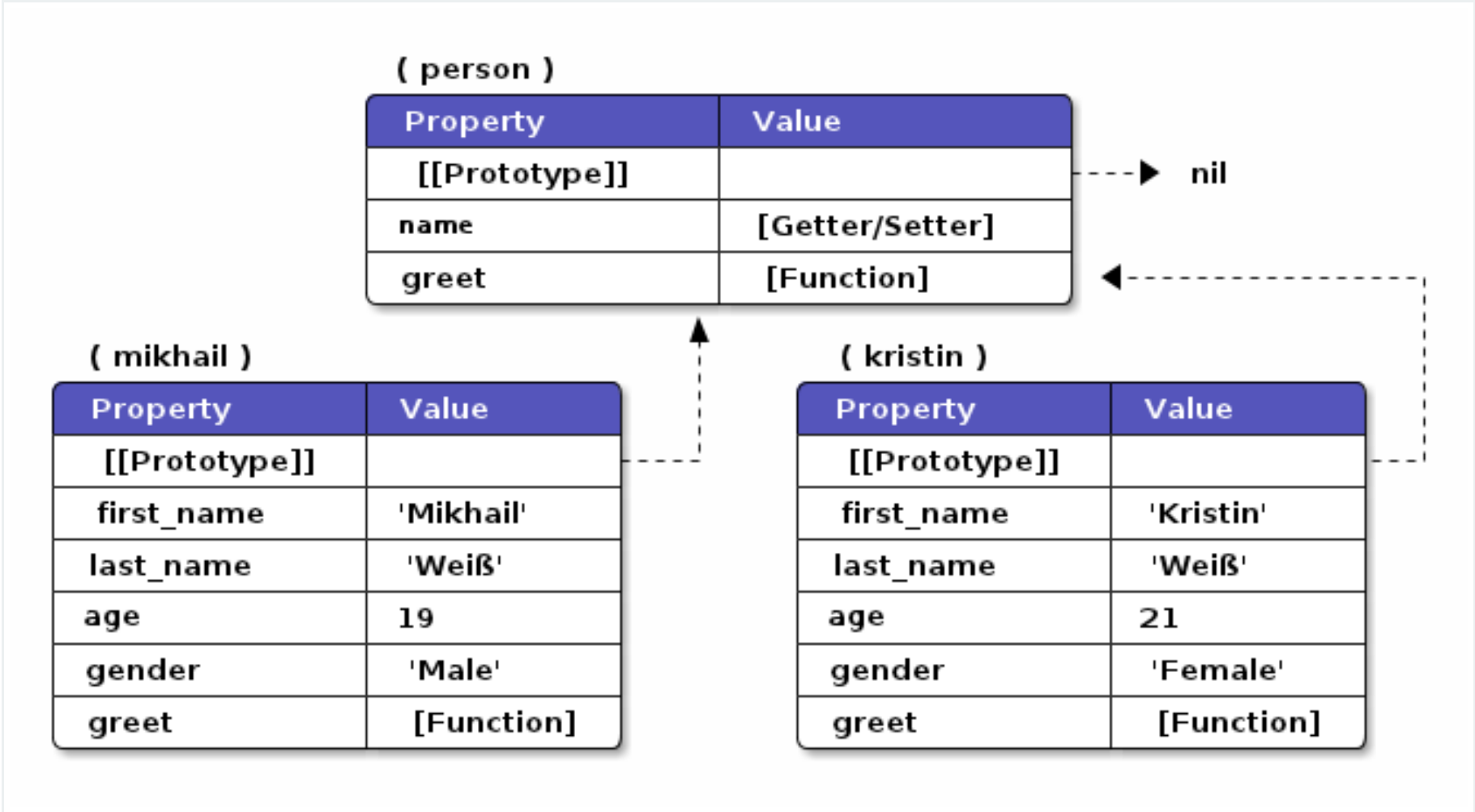
```
1// (person:String) → String
2// Greets the given person
3person.greet = function(person) {
4    return this.name + ': Harro, ' + person + '.'
5}
6
7mikhail.greet('you')
8// => 'Michael White: Harro, you.'
```

3.3. Overriding properties

So, prototypes (that is, inheritance) are used for sharing data with other objects, and it does such in a pretty fast and memory-effective manner too, since you'll always have only one instance of a given piece of data lying around.

Now what if we want to add specialised behaviours, that build upon the data that was shared with the object? Well, we have seen before that objects define their own behaviours by means of properties, so specialised behaviours follow the same principle – you just assign a value to the relevant property.

To better demonstrate it, suppose `Person` implements only a general greeting, and everyone inheriting from `Person` define their own specialised and unique greetings. Also, let's add a new person to our case scenario, so to outline better how objects are extended:



Note that both `mikhail` and `kristin` define their own version of `greet`. In this case, whenever we call the `greet` method on them they'll use their own version of that behaviour, instead of the one that was shared from `person`.

1// Here we set up the greeting for a generic person

```
2
3// (person:String) → String
4// Greets the given person, formally
5person.greet = function(person) {
6    return this.name + ': Hello, ' + (person || 'you')
7}
8
9// And a greeting for our protagonist, Mikhail
10
11// (person:String) → String
12// Greets the given person, like a bro
13mikhail.greet = function(person) {
14    return this.name + ': \'sup, ' + (person || 'dude')
15}
16
17// And define our new protagonist, Kristin
18var kristin = Object.create(person)
19kristin.first_name = 'Kristin'
20kristin.last_name  = 'Weiß'
21kristin.age        = 19
22kristin.gender     = 'Female'
23
24// Alongside with her specific greeting manners
25
26// (person:String) → String
27// Greets the given person, sweetly
28kristin.greet = function(person) {
29    return this.name + ': \'ello, ' + (person || 'sweetie')
30}
31
32// Finally, we test if everything works according to the expected
33
34mikhail.greet(kristin.first_name)
35// => 'Michael White: \'sup, Kristin'
36
37mikhail.greet()
38// => 'Michael White: \'sup, dude'
39
40kristin.greet(mikhail.first_name)
41// => 'Kristin Weiß: \'ello, Michael'
42
```

```
43// And just so we check how cool this [[Prototype]] thing is,
44// let's get Kristin back to the generic behaviour
45
46delete kristin.greet
47// => true
48
49kristin.greet(mikhail.first_name)
50// => 'Kristin Weiß: Hello, Michael'
```

3.4. Mixins

Prototypes allow for behaviour sharing in JavaScript, and although they are undeniably powerful, they aren't quite as powerful as they could be. For one, prototypes only allow that one object inherit from another single object, while extending those behaviours as they see fit.

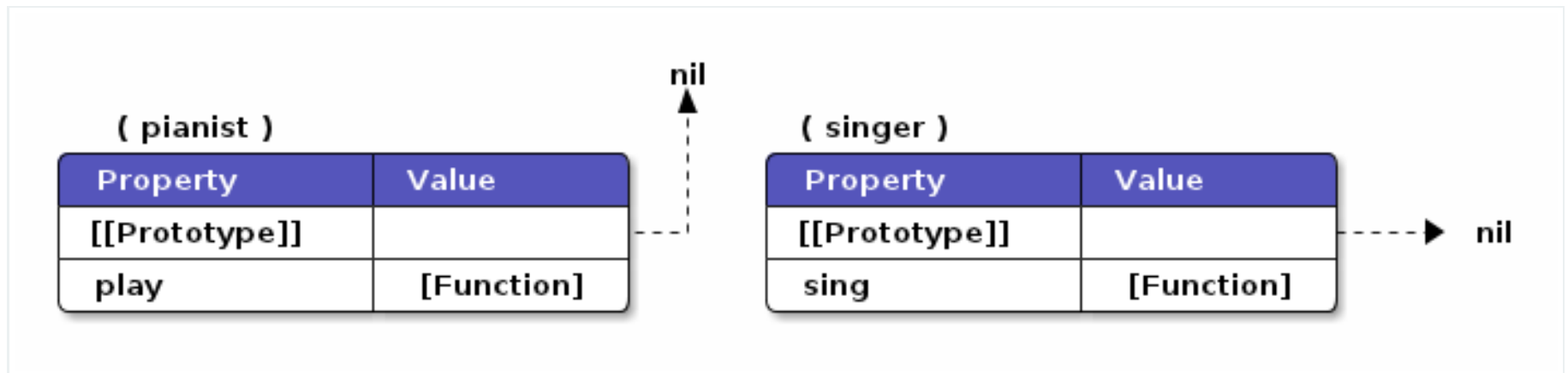
However, this approach quickly kills interesting things like behaviour composition, where we could mix-and-match several objects into one, with all the advantages highlighted in the prototypical inheritance.

Multiple inheritance would also allow the usage of *data-parents* – objects that provide an example state that fulfils the requirements for a given behaviour. Default properties, if you will.

Luckily, since we can define behaviours directly on an object in JavaScript, we can work-around these issues by using mixins – and adding a little overhead at object's creation time.

So, what are mixins anyways? Well, they are parent-less objects. That is, they fully define their own behaviour, and are mostly designed to be incorporated in other objects (although you could use their methods directly).

Continuing with our little protagonists' scenario, let's extend it to add some capabilities to them. Let's say that every person can also be a `pianist` or a `singer`. A given person can have no such abilities, be just a pianist, just a singer or both. This is the kind of case where JavaScript's model of prototypical inheritance falls short, so we're going to cheat a little bit.



For mixins to work, we first need to have a way of combining different objects into a single one. JavaScript doesn't provide this out-of-the box, but we can easily make one by copying all **own** property descriptors, the ones defined directly in the object, rather than inherited, from one object to another.

```
1// Aliases for the rather verbose methods on ES5
2var descriptor = Object.getOwnPropertyDescriptor
3  , properties = Object.getOwnPropertyNames
4  , define_prop = Object.defineProperty
5
6// (target:Object, source:Object) → Object
7// Copies properties from `source` to `target`
8function extend(target, source) {
9  properties(source).forEach(function(key) {
10    define_prop(target, key, descriptor(source, key)) })
11
12  return target
13}
```


Basically, what `extend` does here is taking two objects – a source and a target, – iterating over all properties present on the `source` object, and copying the property descriptors over to `target`. Note that this is a destructive method, meaning that `target` will be modified in-place. It's the cheapest way, though, and usually not a problem.

Now that we have a method for copying properties over, we can start assigning multiple abilities to our objects (`mikhail` e `kristin`):

```
1// A pianist is someone who can `play` the piano
2var pianist = Object.create(null)
3pianist.play = function() {
4    return this.name + ' starts playing the piano.'
5}
6
7// A singer is someone who can `sing`
8var singer = Object.create(null)
9singer.sing = function() {
10    return this.name + ' starts singing.'
11}
12
13// Then we can move on to adding those abilities to
14// our main objects:
15extend(mikhail, pianist)
16mikhail.play()
17// => 'Michael White starts playing the piano.'
18
19// We can see that all that ends up as an own property of
20// mikhail. It is not shared.
21Object.keys(mikhail)
22['first_name', 'last_name', 'age', 'gender', 'greet', 'play']
23
24// Then we can define kristin as a singer
25extend(kristin, singer)
26kristin.sing()
27// => 'Kristin Weiß starts singing.'
28
```

```
29// Mikhail can't sing yet though
30mikhail.sing()
31// => TypeError: Object #<Object> has no method 'sing'
32
33// But mikhail will inherit the `sing` method if we
34// extend the Person prototype with it:
35extend(person, singer)
36
37mikhail.sing()
38// => 'Michael White starts singing.'
```

3.5. Accessing overwritten properties

Now that we're able to inherit properties from other objects and extend the specialised objects to define their own behaviours, we have a little problem: what if we actually wanted to access the parent behaviours that we just overwrote?

JavaScript provides the `Object.getPrototypeOf` function, that returns the `[[Prototype]]` of an object. This way, we have access to all properties defined within the prototype chain of an object. So, accessing a property in the parent of an object is quite simple:

```
1Object.getPrototypeOf(mikhail).name    // same as `person.name`
2// => 'undefined undefined'
3
4// We can assert it's really being called on `person` by
5// giving `person` a `first_name` and `last_name`
6person.first_name = 'Random'
7person.last_name  = 'Person'
8Object.getPrototypeOf(mikhail).name
9// => 'Random Person'
```

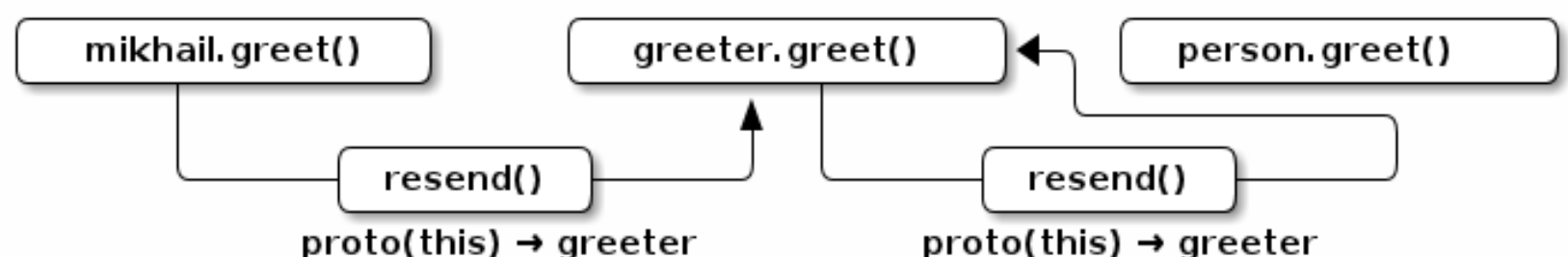
So, a naïve solution for applying a method stored in the `[[Prototype]]` of an object to the current one, would then follow,

quite naturally, by looking the property on the `[[Prototype]]` of `this`:

```
1 var proto = Object.getPrototypeOf
2
3 // (name:String) → String
4 // Greets someone intimately if we know them, otherwise use
5 // the generic greeting
6 mikhail.greet = function(name) {
7     return name == 'Kristin Weiß'? this.name + ': Heya, Kristty'
8     : /* we dunno this guy */ proto(this).greet.call(this,
9 name)
10 }
11 mikhail.greet(kristin.name)
12 // => 'Michael White: Heya, Kristty'
13
14 mikhail.greet('Margareth')
15 // => 'Michael White: Hello, Margareth'
```

This looks all good and well, but there's a little catch: it will enter in endless recursion if you try to apply this approach to more than one parent. This happens because the methods are always applied in the context of the message's first target, making the `[[Prototype]]` lookup resolve always to the same object:

(All methods have `this` as `mikhail`)



The simple solution to this, then, is to make all parent look-ups static, by

passing the object where the current function is stored, rather than the object that the function was applied to.

So, the last example becomes:

```
1 var proto = Object.getPrototypeOf
2
3 // (name:String) → String
4 // Greet someone intimately if we know them, otherwise use
5 // the generic greeting.
6 //
7 // Note that now we explicitly state that the lookup should take
8 // the parent of `mikhail`, so we can be assured the cyclic parent
9 // resolution above won't happen.
10 mikhail.greet = function(name) {
11     return name == 'Kristin Weiß'? this.name + ': Heya, Kristty'
12     : /* we dunno this guy */
13 }
14 proto(mikhail).greet.call(this, name)
15
16 mikhail.greet(kristin.name)
17 // => 'Michael White: Heya, Kristty'
18
19 mikhail.greet('Margareth')
20 // => 'Michael White: Hello, Margareth'
```

Still, this has quite some short-comings. First, since the object is hard-coded in the function, we can't just assign the function to any object and have it just work, as we did up 'till now. The function would always resolve to the parent of `mikhail`, not of the object where it's stored.

Likewise, we can't just apply the function to any object. The function is not generic anymore. Unfortunately, though, making the parent resolution dynamic would require us to pass an additional parameter to every function call, which is something that can't be achieved short of

ugly hacks.

The approach proposed for the next version of JavaScript only solves the first problem, which is the easiest. Here, we'll do the same, by introducing a new way of defining methods. Yes, methods, not generic functions.

Functions that need to access the properties in the `[[Prototype]]` will require an additional information: the object where they are stored. This makes the lookup static, but solves our cyclic lookup problem.

We do this by introducing a new function – `make_method` – which creates a function that passes this information to the target function.

```
// (object:Object, fun:Function) → Function
1 // Creates a method
2 function make_method(object, fun) {
3     return function() { var args
4         args = slice.call(arguments)
5         args.unshift(object)          // insert `object` as first
6 parameter
7         fn.apply(this, args) }
8 }
9
10
11 // Now, all functions that are expected to be used as a method
12 // should remember to reserve the first parameter to the object
13 // where they're stored.
14 //
15 // Note that, however, this is a magical parameter introduced
16 // by the method function, so any function calling the method
17 // should pass only the usual arguments.
18 function message(self, message) { var parent
19     parent = Object.getPrototypeOf(self)
20     if (parent && parent.log)
21         parent.log.call(this, message)
22 }
```

```

23
24     console.log('-- At ' + self.name)
25     console.log(this.name + ': ' + message)
26}
27
28// Here we define a prototype chain C -> B -> A
29var A = Object.create(null)
30A.name = 'A'
31A.log = make_method(A, message)
32
33var B = Object.create(A)
34B.name = 'B'
35B.log = make_method(B, message)
36
37var C = Object.create(B)
38C.name = 'C'
39C.log = make_method(C, message)
40
41// And we can test if it works by calling the methods:
42A.log('foo')
43// => '-- At A'
44// => 'A: foo'
45
46B.log('foo')
47// => '-- At A'
48// => 'B: foo'
49// => '-- At B'
50// => 'B: foo'
51
52C.log('foo')
53// => '-- At A'
54// => 'C: foo'
55// => '-- At B'
56// => 'C: foo'
57// => '-- At C'
    // => 'C: foo'

```

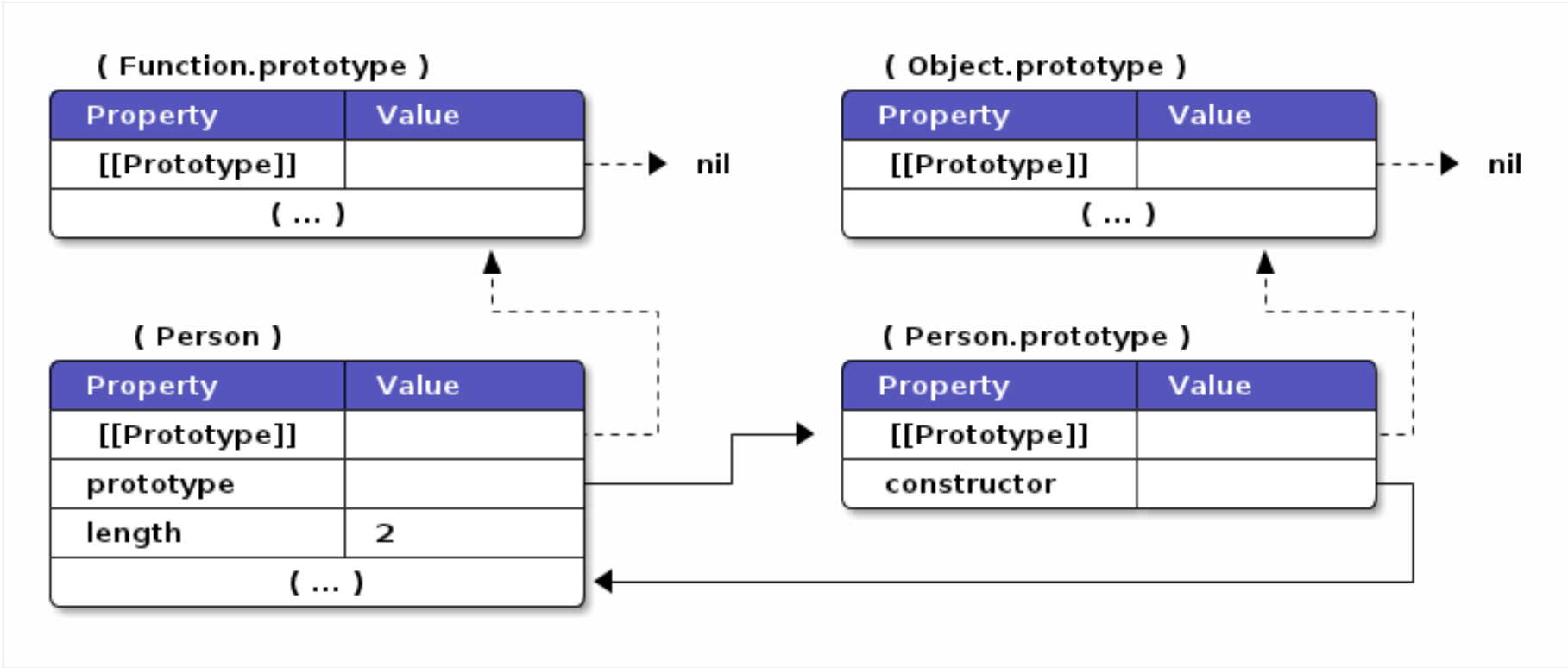
4. Constructors

Constructor functions are the old pattern for creating objects in JavaScript, which couple inheritance with initialisation in an imperative manner.

Constructor functions **are not**, however, a special construct in the language. Any simple function can be used as a constructor function; just like `this`, it all depends on how the function is called.

So, what's it about constructor functions, really? Well, every function object in JavaScript automatically gets a `prototype` property, that is a simple object with a `constructor` property pointing back to the constructor function. And this object is used to determine the `[[Prototype]]` of instances created with that constructor function.

The following diagram shows the objects for the constructor function `function Person(first_name, last_name):`



4.1. The `new` magic

The `prototype` *per se* is not a special property, however it gains special

meaning when a constructor function is used in conjunction with the `new` statement. As I said before, in this case the `prototype` property of the constructor function is used to provide the `[[Prototype]]` of the instance.

```
1// Constructs a new Person
2function Person(first_name, last_name) {
3    // If the function is called with `new`, as we expect
4    // `this` here will be the freshly created object
5    // with the [[Prototype]] set to Person.prototype
6    //
7    // Of course, if someone omits new when calling the
8    // function, the usual resolution of `this` – as
9    // explained before – will take place.
10   this.first_name = first_name
11   this.last_name  = last_name
12}
13
14// (person:String) → String
15// Greets the given person
16Person.prototype.greet = function(person) {
17   return this.name + ': Harro, ' + person + '.'
18}
19
20var person = new Person('Mikhail', 'Weiß')
21
22
23// We could de-sugar the constructor pattern in the following
24// Taking into account that `Person` here means the `prototype`
25// property of the `Person` constructor.
26var Person = Object.create(Object.prototype)
27
28// (person:String) → String
29// Greets the given person
30Person.greet = function(person) {
31   return this.name + ': Harro, ' + person + '.'
32}
33
34// Here's what the constructor does when called with `new`
```

```
35var person = Object.create(Person)
36person.first_name = 'Mikhail'
37person.last_name = 'Weiß'
```

When a function is called with the `new` statement, the following magic happens:

1. Create a fresh `Object`, inheriting from `Object.prototype`, say `{ }`
2. Set the `[[Prototype]]` internal property of the new object to point to the constructor's `prototype` property, so it inherits those behaviours.
3. Call the constructor in the context of this fresh object, such that `this` inside the constructor will be the fresh object, and pass any parameters given to the function.
4. If the function returns an `Object`, make that be the return value of the function.
5. Otherwise, return the fresh object.

This means that the resulting value of calling a `function` with the `new` operator is not necessarily the object that was created. A function is free to return any other `Object` value as it sees fit. This is an interesting and – to a certain extent – powerful behaviour, but also a confusing one for many newcomers:

```
1function Foo() {
2    this.foo = 'bar'
3}
4
5new Foo()
```

```
6// => { foo: 'bar' }
7
8
9function Foo() {
10    this.foo = 'bar'
11    return Foo
12}
13
14new Foo()
15// => [Function: Foo]
```

4.2. Inheritance with constructors

We've covered inheritance with plain objects through `Object.create`, inheritance with constructors follow quite naturally from there, the difference being that instead of the main actor being the target of the inheritance (the constructor function, in this case), the `prototype` property is:

```
// new Person (first_name:String, last_name:String)
// Initialises a Person object
function Person(first_name, last_name) {
1    this.first_name = first_name
2    this.last_name  = last_name
3}
4
5// Defines the `name` getter/setter
6Object.defineProperty(Person.prototype, 'name', { get:
7get_full_name
8
9set_full_name
10
11true
12
13true })
14
15// (person:String) → String
16
```

```

17// Greets the given person
18Person.prototype.greet = function(person) {
19    return this.name + ': Hello, ' + (person || 'you')
20}
21
22
23var proto = Object.getPrototypeOf
24
25// new Mikhail (age:Number, gender:String)
26function Mikhail(age, gender) {
27    // Find the parent of this object and invoke its constructor
28    // with the current this. We could have used:
29    //    Person.call(this, 'Mikhail', 'Weiß')
30    // But we'd lose some flexibility with that.
31    proto(Mikhail.prototype).constructor.call(this, 'Mikhail',
32'Weiß')
33}
34
35// Inherits the properties from Person.prototype
36Mikhail.prototype = Object.create(Person.prototype)
37
38// Resets the `constructor` property of the prototype object
39Mikhail.prototype.constructor = Mikhail
40
41// (person:String) → String
42Mikhail.prototype.greet = function(person) {
43    return this.name + ': \'sup, ' + (person || 'dude')
44}
45
46
47// Instances are created with the `new` operator, as previously
48// discussed:
49    var mikhail = new Mikhail(19, 'Male')
50    mikhail.greet('Kristin')
51    // => 'Mikhail Weiß: \'sup, Kristin'

```

5. Considerations and compatibility

The functions and concepts presented up until now assumed that the

code would be running in an ECMAScript 5 environment, since the new additions make prototypical inheritance more natural, without the initialisation and inheritance coupling provided by constructor functions.

However, obviously this means that code using these functions will not work everywhere. [@kangax](#) has a most awesome [compatibility table](#) for the implementations that follow ECMAScript 5.

This section provides fallbacks to some of the functionality, and point to libraries that implement these fallbacks so you don't get to reinvent the wheel. Note that this section only exists to highlight how the functionality works, and how the core part of those behaviours could be reproduced in legacy code, it's not meant to provide ready-to-use fallbacks. Use libraries for that :3

5.1. Creating objects

In ECMAScript 5 we have got `Object.create` to handle inheritance, but constructor functions can also be used to set the `[[Prototype]]` link for the constructed object – which is what we're interested about.

A `clone` function could be defined such that it would create a new object based on the given prototype:

```
1 // (proto:Object) → Object
2 // Constructs an object and sets the [[Prototype]] to `proto`.
3 function clone(proto) {
4     function Dummy() { }
5
6     Dummy.prototype = proto
7     Dummy.prototype.constructor = Dummy
8 }
```

```
9     return new Dummy()
10}
11
12var mikhail = clone(person)
13// Equivalent to `var mikhail = Object.create(person)'
```

5.2. Defining properties

`Object.defineProperty` and its batch cousin

`Object.defineProperties` are also new additions, and they allow properties to be defined with internal tags, like `writable`, `configurable` and `enumerable`. It's not possible to get this behaviour in the older versions of the language.

All properties defined otherwise will inevitably have `writable`, `configurable` and `enumerable` set to true, which is usually not really that much of a problem – still, not compatible with full ES5 code.

In regards of getters and setters, they are supported to a certain extent with non-standard syntax – the `__defineGetter__` and `__defineSetter__` methods, – but are also not available everywhere. Most notably, such methods have never been present in IE.

```
// (target:Object, key:String, descriptor:Object) → Object
1// Defines a property in the target object.
2// Getters and Setters are handled through the fallback
3// calls, whereas values are set directly. Tags are
4// ignored.
5function defineProperty(target, key, descriptor) {
6    if (descriptor.value)
7        target[key] = descriptor.value
8    else {
9        descriptor.get && target.__defineGetter__(key,
10descriptor.get)
```

```

11         descriptor.set && target.__defineSetter__(key,
12 descriptor.set) }
13
14     return target
15}
16
17
18var x = { }
19defineProperty(x, 'foo', { value: 'bar' })
20defineProperty(x, 'bar', { get: function() { return this.foo }
21                        , set: function(v){ this.foo = v      }})
22
23x.foo
24// => 'bar'
25
26x.bar
27// => 'bar'
28
29x.bar = 'foo'
30x.foo
31// => 'foo'
32
33x.bar
34// => 'foo'

```

5.3. Listing properties

We have seen how it's possible to list the properties of an object with `Object.getOwnPropertyNames`, and list only the enumerable properties through `Object.keys`. Well, prior to ECMAScript 5, listing the enumerable properties is the only thing one can do.

This is achieved through the `for..in` statement, which iterates through all the enumerable properties of an object, either directly set in the object, or in the prototype chain.

`Object.prototype.hasOwnProperty` may be used to filter the

properties to include only the ones set directly in the object.

```
1// (object:Object) → Array
2// Lists all the own enumerable properties of an object
3function keys(object) { var result, key
4    result = []
5    for (key in object)
6        if (object.hasOwnProperty(key)) result.push(key)
7
8    return result
9}
10
11// Taking the mikhail object whose [[Prototype]] is person...
12keys(mikhail)
13// => [ 'first_name', 'last_name', 'age', 'gender' ]
14
15keys(person)
16// => [ 'greet', 'name' ]
```

5.4. Bound methods

Bound methods in JavaScript do much more than just assert the value of `this` inside a function, they can also be used for partial function applications and behave slightly different when called as a constructor. For this, we'll just focus on the first two.

Basically, when calling the `bind` method of a function, we're creating a new function object that has a defined `thisObject` and perhaps a defined initial list of arguments. This can be just as well achieved with a closure to store the given state, and a explicit function application, through the `apply` method.

```
1var slice = [].slice
2
3// (fun:Function, bound_this:Object, args...) → Function
```

```

4// --> (args...) → *mixed*
5// Creates a bound method from the function `fn`
6function bind(fn, bound_this) { var bound_args
7    bound_args = slice.call(arguments, 2)
8    return function() { var args
9        args = bound_args.concat(slice.call(arguments))
10        return fn.apply(bound_this, args) }
11}

```

5.5. Getting the `[[Prototype]]`

For accessing overridden properties, we need to get a reference to the `[[Prototype]]`. In environments that expose such link (like Firefox's *SpiderMonkey* or Chrome's V8), it's easy and reliable:

```

1function proto(object) {
2    return object?      object.__proto__
3        : /* not object? */ null
4}

```

However, in environments that don't expose the `[[Prototype]]` link, things aren't quite as reliable. The only way of getting the prototype of an object, in this case, would be by the constructor's `prototype` property, but we can only access that from the object given the `constructor` property is kept intact.

A fallback covering most cases would look like this:

```

1function proto(object) {
2    return !object?      null
3        : '.__proto__' in object? object.__proto__
4        : /* not exposed? */ object.constructor.prototype
5}

```

Note that the actual `Object.getPrototypeOf` throws a `TypeError` when you pass something that is not an object to it.

5.6. Libraries that provide fallbacks

[ES5-shim](#) attempts to implement fallbacks for ECMAScript 5 functionality that can be done in pure JavaScript, whilst adding minimal support for the other ones. It's important to note, however, that the fallbacks are intended to provide equivalent functionality that is close to the ones defined in the specs, it's not guaranteed that they will work exactly the same way.

To quote the [README](#):

"As closely as possible to ES5" is not very close. Many of these shims are intended only to allow code to be written to ES5 without causing run-time errors in older engines. In many cases, this means that these shims cause many ES5 methods to silently fail. Decide carefully whether this is what you want.

6. Wrapping it up

The object orientation model chosen for JavaScript is definitely one of the things that makes the language expressive and powerful, however the really poor semantics from the before-ES5 age quite killed all the fun about it.

With ECMAScript 5, we have got better ways to deal with objects and inheritance, but most of the API is pretty verbose and awkward to use out of the box, so abstracting them is the only good way of exploring all

the power of the first-class inheritance model provided by the language.

Once you dwell on the depths of JavaScript's prototypical object orientation, however, you will find it lacking on aspects that would otherwise seem like the obvious thing to do – like multiple inheritance and message resending, but also basic features like an easier object extension functionality.

Luckily most of these issues manage to have a solution, albeit not necessarily a satisfactory one in some cases – i.e.: manual mixins. Being able to reproduce semantics that are not provided straight away on the language by patterns leveraging the built-in constructs is an important part of the language, and this all is made easier because of the way functions are treated in JavaScript.

7. Things worth reading up next

BRENDAN EICH'S "PROTOTYPICAL VS CLOSURE" RANT

Although not really a reading, this particular podcast from Brendan Eich is a must listen for anyone working with object oriented JavaScript. it delves on the performance of engines in regards of object construction, highlighting how the prototypical pattern stands against the [Closure pattern](#), and discussing the specifics of how browsers handle prototypical code so they run **fast**.

ORGANIZING PROGRAMS WITHOUT CLASSES (*PAPER*)

Albeit not specific to JavaScript, this white-paper dwells on how the structuring of programs differ from the classical object orientation approach to the prototypical take on the subject. It provides lots of

[Self](#) code to go with it, but they are more or less easily translated to JavaScript code.

8. Acknowledgements

Thanks to [@hughfdjackson](#), Alex and Taylor for the additional revision of the article.

Thanks to `--` for pointing out (in the comments) that properties expect an `IdentifierName`, rather than a plain `Identifier`, which would allow reserved words (like `foo.class` or `foo.null`) to be used unquoted.

Footnotes:

¹: Some implementations have magical names, like `__proto__`, which may yield undesired and unwanted results when set. For example, the `__proto__` property is used to define the parent of an object in some implementations. As such, you wouldn't be able to set a string or number to that.

²: While an `IdentifierName` also allows reserved words, you should be aware that ECMAScript engines that aren't fully compliant with the ECMAScript 5 specs **may** choke on reserved words when they're used for property access unquoted.

³: It should be noted that, while ECMAScript-defined native objects don't throw an error when you try to access a non-existing property, it's not guaranteed that the same will hold true for a host object (i.e. an

object defined by the engine implementor). After all, host object semantics are not defined, they are dependent on the particular run-time implementation.

⁴ Some engines **do** expose the `[[Prototype]]` slot, usually through a property like `__proto__`, however no such thing is described in the specifications for the language, so it's recommended that you avoid using it, unless you're well aware that all platforms you code must run on will have such means of setting the `[[Prototype]]` object directly. It should also be noted that messing with the prototype chain might defeat all look-up optimisations in the JS engine.

Quil really hates getters and setters, and really loves message passing. She often doesn't make much sense. You can contact her on [Twitter](#) or [Email](#).



Made with ❤️ by Quil.
Fonts feat. [Google Fonts](#) & [Entypo](#)



This work is licensed under a [Creative Commons Attribution 3.0 Unported License](#).