The Node Craftsman Book

An advanced Node.js tutorial

Leanpub

Manuel Kiessling

The Node Craftsman Book

An advanced Node.js tutorial

Manuel Kiessling

This book is for sale at http://leanpub.com/nodecraftsman

This version was published on 2017-11-18

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Manuel Kiessling by spreading the word about this book on Twitter!

The suggested hashtag for this book is #nodecraftsman.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#nodecraftsman

Also By Manuel Kiessling

The Node Beginner Book

Node入门

El Libro Principiante de Node

Livro do Iniciante em Node

Beginning Mobile App Development with React Native

For Nora, Aaron and Melinda.

Contents

Sample chapter: Object-oriented JavaScript **1**
Blueprints versus finger-pointing 1
A classless society 2
Creating objects 2
Object-orientation, prototyping, and inheritance 10
A classless society, revisited 11
Summary 13

Sample chapter: Object-oriented JavaScript

Let's talk about object-orientation and inheritance in JavaScript.

The good news is that it's actually quite simple, but the bad news is that it works completely different than object-orientation in languages like C++, Java, Ruby, Python or PHP, making it not-quite-so simple to understand.

But fear not, we are going to take it step by step.

Blueprints versus finger-pointing

Let's start by looking at how "typical" object-oriented languages actually create objects.

We are going to talk about an object called *myCar*. *myCar* is our bits-and-bytes representation of an incredibly simplified real world car. It could have attributes like *color* and *weight*, and methods like *drive* and *honk*.

In a real application, *myCar* could be used to represent the car in a racing game - but we are going to completely ignore the context of this object, because we will talk about the nature and usage of this object in a more abstract way.

If you would want to use this myCar object in, say, Java, you need to define the blueprint of this specific object first - this is what Java and most other object-oriented languages call a *class*.

If you want to create the object *myCar*, you tell Java to "build a new object after the specification that is laid out in the class Car".

The newly built object shares certain aspects with its blueprint. If you call the method honk on your object, like so:

```
myCar.honk();
```

then the Java VM will go to the class of *myCar* and look up which code it actually needs to execute, which is defined in the *honk* method of class *Car*.

Ok, nothing shockingly new here. Enter JavaScript.

A classless society

JavaScript does not have classes. But as in other languages, we would like to tell the interpreter that it should build our *myCar* object following a certain pattern or schema or blueprint - it would be quite tedious to create every *car* object from scratch, "manually" giving it the attributes and methods it needs every time we build it.

If we were to create 30 *car* objects based on the *Car* class in Java, this object-class relationship provides us with 30 cars that are able to drive and honk without us having to write 30 *drive* and *honk* methods.

How is this achieved in JavaScript? Instead of an object-class relationship, there is an object-object relationship.

Where in Java our *myCar*, asked to *honk*, says "go look at this class over there, which is my *blueprint*, to find the code you need", JavaScript says "go look at that other object over there, which is my *prototype*, it has the code you are looking for".

Building objects via an object-object relationship is called Prototype-based programming, versus Class-based programming used in more traditional languages like Java.

Both are perfectly valid implementations of the object-oriented programming paradigm - it's just two different approaches.

Creating objects

Let's dive into code a bit, shall we? How could we set up our code in order to allow us to create our *myCar* object, ending up with an object that is a Car and can therefore *honk* and *drive*?

Well, in the most simple sense, we can create our object completely from scratch, or ex nihilo if you prefer the boaster expression.

It works like this:

```
1  var myCar = {};
2
3  myCar.honk = function() {
4    console.log('honk honk');
5  };
6
7  myCar.drive = function() {
8    console.log('vrooom...');
9  };
```

This gives us an object called *myCar* that is able to *honk* and *drive*:

```
myCar.honk();   // outputs "honk honk"
myCar.drive();  // outputs "vrooom..."
```

However, if we were to create 30 cars this way, we would end up defining the honk and drive behaviour of every single one, something we said we want to avoid.

In real life, if we made a living out of creating, say, pencils, and we don't want to create every pencil individually by hand, then we would consider building a pencil-making machine, and have this machine create the pencils for us.

After all, that's what we implicitly do in a class-based language like Java - by defining a class Car, we get the car-maker for free:

```
Car myCar = new Car();
```

will build the *myCar* object for us based on the *Car* blueprint. Using the *new* keyword does all the magic for us.

JavaScript, however, leaves the responsibility of building an object creator to us. Furthermore, it gives us a lot of freedom regarding the way we actually build our objects.

In the most simple case, we can write a function which creates "plain" objects that are exactly like our "ex nihilo" object, and that don't really share any behaviour - they just happen to roll out of the factory with the same behaviour copied onto every single one, if you want so.

Or, we can write a special kind of function that not only creates our objects, but also does some behind-the-scenes magic which links the created objects with their creator. This allows for a true sharing of behaviour: functions that are available on all created objects point to a single implementation. If this function implementation changes after objects have been created, which is possible in JavaScript, the behaviour of all objects sharing the function will change accordingly.

Let's examine all possible ways of creating objects in detail.

Using a simple function to create plain objects

In our first example, we created a plain *myCar* object out of thin air - we can simply wrap the creation code into a function, which gives us a very basic object creator:

```
var makeCar = function() {
  var newCar = {};
  newCar.honk = function() {
    console.log('honk honk');
  };
};
```

For the sake of brevity, the drive function has been omitted.

We can then use this function to mass-produce cars:

```
var makeCar = function() {
  var newCar = {}
  newCar.honk = function() {
    console.log('honk honk');
  };
  return newCar;
};

myCar1 = makeCar();
myCar2 = makeCar();
myCar3 = makeCar();
```

One downside of this approach is efficiency: for every *myCar* object that is created, a new *honk* function is created and attached - creating 1,000 objects means that the JavaScript interpreter has to allocate memory for 1,000 functions, although they all implement the same behaviour. This results in an unnecessarily high memory footprint of the application.

Secondly, this approach deprives us of some interesting opportunities. These *myCar* objects don't share anything - they were built by the same creator function, but are completely independent from each other.

It's really like with real cars from a real car factory: They all look the same, but once they leave the assembly line, they are totally independent. If the manufacturer should decide that pushing the horn on already produced cars should result in a different type of honk, all cars would have to be returned to the factory and modified.

In the virtual universe of JavaScript, we are not bound to such limits. By creating objects in a more sophisticated way, we are able to magically change the behaviour of all created objects at once.

Using a constructor function to create objects

In JavaScript, the entities that create objects with shared behaviour are functions which are called in a special way. These special functions are called *constructors*.

Let's create a constructor for cars. We are going to call this function *Car*, with a capital *C*, which is common practice to indicate that this function is a constructor.

In a way, this makes the constructor function a class, because it does some of the things a class (with a constructor method) does in a traditional OOP language. However, the approach is not identical, which is why constructor functions are often called *pseudo-classes* in JavaScript. I will simply call them classes or constructor functions.

Because we are going to encounter two new concepts that are both necessary for shared object behaviour to work, we are going to approach the final solution in two steps.

Step one is to recreate the previous solution (where a common function spilled out independent car objects), but this time using a constructor:

```
var Car = function() {
  this.honk = function() {
    console.log('honk honk');
  };
};
```

When this function is called using the *new* keyword, like so:

```
var myCar = new Car();
```

it implicitly returns a newly created object with the *honk* function attached.

Using *this* and *new* makes the explicit creation and return of the new object unnecessary - it is created and returned "behind the scenes" (i.e., the *new* keyword is what creates the new, "invisible" object, and secretly passes it to the *Car* function as its *this* variable).

You can think of the mechanism at work a bit like in this pseudo-code:

```
// Pseudo-code, for illustration only!

var Car = function(this) {
  this.honk = function() {
    console.log('honk honk');
  };
  return this;
};

var newObject = {};
var myCar = Car(newObject);
```

As said, this is more or less like our previous solution - we don't have to create every car object manually, but we still cannot modify the *honk* behaviour only once and have this change reflected in all created cars.

But we laid the first cornerstone for it. By using a constructor, all objects received a special property that links them to their constructor:

```
var Car = function() {
  this.honk = function() {
    console.log('honk honk');
  };
};

var myCar1 = new Car();
var myCar2 = new Car();

console.log(myCar1.constructor);  // outputs [Function: Car]
console.log(myCar2.constructor);  // outputs [Function: Car]
```

All created *myCars* are linked to the *Car* constructor. This is what actually makes them a *class* of related objects, and not just a bunch of objects that happen to have similar names and identical functions.

Now we have finally reached the moment to get back to the mysterious prototype we talked about in the introduction.

Using prototyping to efficiently share behaviour between objects

As stated there, while in class-based programming the class is the place to put functions that all objects will share, in prototype-based programming, the place to put these functions is the object which acts as the prototype for our objects at hand.

But where is the object that is the prototype of our *myCar* objects - we didn't create one!

It has been implicitly created for us, and is assigned to the *Car.prototype* property (in case you wondered, JavaScript functions are objects too, and they therefore have properties).

Here is the key to sharing functions between objects: Whenever we call a function on an object, the JavaScript interpreter tries to find that function within the queried object. But if it doesn't find the function within the object itself, it asks the object for the pointer to its prototype, then goes to the prototype, and asks for the function there. If it is found, it is then executed.

This means that we can create *myCar* objects without any functions, create the *honk* function in their prototype, and end up having *myCar* objects that know how to honk - because everytime the interpreter tries to execute the *honk* function on one of the myCar objects, it will be redirected to the prototype, and execute the *honk* function which is defined there.

Here is how this setup can be achieved:

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

Car.prototype.honk = function() {
  console.log('honk honk');
};

var myCar1 = new Car();
var myCar2 = new Car();

myCar1.honk();  // executes Car.prototype.honk() and outputs "honk honk"
myCar2.honk();  // executes Car.prototype.honk() and outputs "honk honk"
```

Our constructor is now empty, because for our very simple cars, no additional setup is necessary.

Because both *myCar*s are created through this constructor, their prototype points to *Car.prototype* - executing *myCar1.honk()* and *myCar2.honk()* always results in *Car.prototype.honk()* being executed.

Let's see what this enables us to do. In JavaScript, objects can be changed at runtime. This holds true for prototypes, too. Which is why we can change the *honk* behaviour of all our cars even after they have been created:

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

Car.prototype.honk = function() {
  console.log('honk honk');
};

var myCar1 = new Car();
var myCar2 = new Car();

myCar1.honk();  // executes Car.prototype.honk() and outputs "honk honk"
myCar2.honk();  // executes Car.prototype.honk() and outputs "honk honk"

Car.prototype.honk = function() {
  console.log('meep meep');
};

myCar1.honk();  // executes Car.prototype.honk() and outputs "meep meep"
myCar2.honk();  // executes Car.prototype.honk() and outputs "meep meep"
```

Of course, we can also add additional functions at runtime:

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

Car.prototype.honk = function() {
  console.log('honk honk');
};

var myCar1 = new Car();
var myCar2 = new Car();

Car.prototype.drive = function() {
  console.log('vrooom...');
};

myCar1.drive();  // executes Car.prototype.drive() and outputs "vrooom..."
myCar2.drive();  // executes Car.prototype.drive() and outputs "vrooom..."
```

But we could even decide to treat only one of our cars differently:

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

Car.prototype.honk = function() {
  console.log('honk honk');
};

var myCar1 = new Car();
var myCar2 = new Car();

myCar1.honk();  // executes Car.prototype.honk() and outputs "honk honk"
myCar2.honk();  // executes Car.prototype.honk() and outputs "honk honk"

myCar2.honk = function() {
  console.log('meep meep');
};

myCar1.honk();  // executes Car.prototype.honk() and outputs "honk honk"
myCar2.honk();  // executes myCar2.honk() and outputs "meep meep"
```

It's important to understand what happens behind the scenes in this example. As we have seen, when calling a function on an object, the interpreter follows a certain path to find the actual location of that function.

While for *myCar1*, there still is no *honk* function within that object itself, that no longer holds true for *myCar2*. When the interpreter calls *myCar2.honk()*, there now is a function within *myCar2* itself.

Therefore, the interpreter no longer follows the path to the prototype of *myCar2*, and executes the function within *myCar2* instead.

That's one of the major differences to class-based programming: while objects are relatively "rigid" e.g. in Java, where the structure of an object cannot be changed at runtime, in JavaScript, the prototype-based approach links objects of a certain class more loosely together, which allows to change the structure of objects at any time.

Also, note how sharing functions through the constructor's prototype is way more efficient than creating objects that all carry their own functions, even if they are identical. As previously stated, the engine doesn't know that these functions are meant to be identical, and it has to allocate memory for every function in every object. This is no longer true when sharing functions through a common prototype - the function in question is placed in memory exactly once, and no matter how many *myCar* objects we create, they don't carry the function themselves, they only refer to their constructor, in whose prototype the function is found.

To give you an idea of what this difference can mean, here is a very simple comparison. The first example creates 1,000,000 objects that all have the function directly attached to them:

```
var C = function() {
  this.f = function(foo) {
    console.log(foo);
  };
};

var a = [];
for (var i = 0; i < 1000000; i++) {
  a.push(new C());
}
```

In Google Chrome, this results in a heap snapshot size of 328 MB. Here is the same example, but now the function is shared through the constructor's prototype:

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

C.prototype.f = function(foo) {
  console.log(foo);
};

var a = [];
for (var i = 0; i < 1000000; i++) {
  a.push(new C());
}
```

This time, the size of the heap snapshot is only 17 MB, i.e., only about 5% of the non-efficient solution.

Object-orientation, prototyping, and inheritance

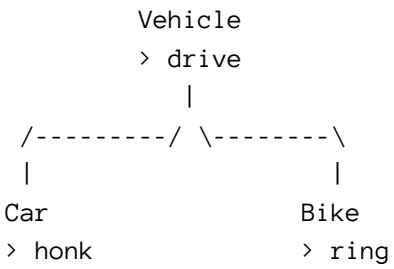
So far, we haven't talked about inheritance in JavaScript, so let's do this now.

It's useful to share behaviour within a certain class of objects, but there are cases where we would like to share behaviour between different, but similar classes of objects.

Imagine our virtual world not only had cars, but also bikes. Both drive, but where a car has a horn, a bike has a bell.

Being able to drive makes both objects vehicles, but not sharing the *honk* and *ring* behaviour distinguishes them.

We could illustrate their shared and local behaviour as well as their relationship to each other as follows:



Designing this relationship in a class-based language like Java is straightforward: We would define a class *Vehicle* with a method *drive*, and two classes *Car* and *Bike* which both extend the *Vehicle* class, and implement a *honk* and a *ring* method, respectively.

This would make the *car* as well as *bike* objects inherit the *drive* behaviour through the inheritance of their classes.

How does this work in JavaScript, where we don't have classes, but prototypes?

Let's look at an example first, and then dissect it. To keep the code short for now, let's only start with a car that inherits from a vehicle:

```
1  var Vehicle = function() {};
2
3  Vehicle.prototype.drive = function() {
4    console.log('vrooom...');
5  };
6
7
8  var Car = function() {};
9
```

```
Car.prototype = new Vehicle();

Car.prototype.honk = function() {
  console.log('honk honk');
};


var myCar = new Car();

myCar.honk();   // outputs "honk honk"
myCar.drive();  // outputs "vrooom..."
```

In JavaScript, inheritance runs through a chain of prototypes.

The prototype of the *Car* constructor is set to a newly created vehicle object, which establishes the link structure that allows the interpreter to look for methods in "parent" objects.

The prototype of the *Vehicle* constructor has a function *drive*. Here is what happens when the *myCar* object is asked to *drive()*:

- The interpreter looks for a *drive* method within the *myCar* object, which does not exist
- The interpreter then asks the *myCar* object for its prototype, which is the prototype of its constructor *Car*
- When looking at *Car.prototype*, the interpreter sees a *vehicle* object which has a function *honk* attached, but no *drive* function
- Thus, the interpreter now asks this *vehicle* object for its prototype, which is the prototype of its constructor *Vehicle*
- When looking at *Vehicle.prototype*, the interpreter sees an object which has a *drive* function attached - the interpreter now knows which code implements the *myCar.drive()* behaviour, and executes it

A classless society, revisited

We just learned how to emulate the traditional OOP inheritance mechanism. But it's important to note that in JavaScript, that is only one valid approach to create objects that are related to each other.

It was Douglas Crockford who came up with another clever solution, which allows objects to inherit from each other directly. It's a native part of JavaScript by now - it's the *Object.create()* function, and it works like this:

```
Object.create = function(o) {
  var F = function() {};
  F.prototype = o;
  return new F();
};
```

We learned enough now to understand what's going on. Let's analyze an example:

```
var vehicle = {};
vehicle.drive = function () {
  console.log('vrooom...');
};

var car = Object.create(vehicle);
car.honk = function() {
  console.log('honk honk');
};

var myCar = Object.create(car);

myCar.honk();   // outputs "honk honk"
myCar.drive();  // outputs "vrooom..."
```

While being more concise and expressive, this code achieves exactly the same behaviour, without the need to write dedicated constructors and attaching functions to their prototype. As you can see, *Object.create()* handles both behind the scenes, on the fly. A temporary constructor is created, its prototype is set to the object that serves as the role model for our new object, and a new object is created from this setup.

Conceptually, this is really the same as in the previous example where we defined that *Car.prototype* shall be a *new Vehicle()*;.

But wait! We created the functions *drive* and *honk* within our objects, not on their prototypes - that's memory-inefficient!

Well, in this case, it's actually not. Let's see why:

```
var vehicle = {};
vehicle.drive = function () {
  console.log('vrooom...');
};

var car = Object.create(vehicle);
car.honk = function() {
  console.log('honk honk');
};

var myVehicle = Object.create(vehicle);
var myCar1 = Object.create(car);
var myCar2 = Object.create(car);

myCar1.honk();  // outputs "honk honk"
myCar2.honk();  // outputs "honk honk"

myVehicle.drive();  // outputs "vrooom..."
myCar1.drive();     // outputs "vrooom..."
myCar2.drive();     // outputs "vrooom..."
```

We have now created a total of 5 objects, but how often do the *honk* and *drive* methods exist in memory? Well, how often have they been defined? Just once - and therefore, this solution is basically as efficient as the one where we built the inheritance manually. Let's look at the numbers:

```
var c = {};
c.f = function(foo) {
  console.log(foo);
};

var a = [];
for (var i = 0; i < 1000000; i++) {
  a.push(Object.create(c));
}
```

Turns out, it's not *exactly* identical - we end up with a heap snapshot size of 40 MB, thus there seems to be some overhead involved. However, in exchange for much better code, this is probably more than worth it.

Summary

By now, it's probably clear what the main difference between classical OOP languages and JavaScript is, conceptually: While classical languages like Java provide *one* way to manage object creation and

behaviour sharing (through classes and inheritance), and this way is enforced by the language and "baked in", JavaScript starts at a slightly lower level and provides building blocks that allow us to create several different mechanisms for this.

Whether you decide to use these building blocks to recreate the traditional class-based pattern, let your objects inherit from each other directly, with the concept of classes getting in the way, or if you don't use the object-oriented paradigm at all and just solve the problem at hand with pure functional code: JavaScript gives you the freedom to choose the best methodology for any situation.