JavaScript Unlocked

JavaScript stands astride the world like a colossus. Having conquered web development, it now advances into new areas such as server scripting, desktop and mobile development, game scripting, and more. This all-practical guide is packed with code recipes to help you unlock the full potential of JavaScript.

Start by diving right into the core of JavaScript, with powerful user techniques for getting better maintainability and performance from the basic building blocks of your code. Get to grips with modular programming to bring real power to the browser, master client-side JavaScript scripting without jQuery or other frameworks, and discover the full potential of asynchronous coding. Do great things with HTML5 APIs, including building your first web component.

With every task demonstrated in both classic ES5 JavaScript and the next generation ES6-7 versions of the language, whether read cover-to-cover or dipped into for specific recipes, JavaScript Unlocked is your essential guide for pushing JavaScript to its limits.

Who this book is written for

JavaScript Unlocked is for those JavaScript developers who want to see just how far they can push their favourite language, using practical insight and techniques.

What you will learn from this book

- Improve the readability of your code by unleashing the latent potential of JavaScript language
- Understand existing JavaScript collections such as arrays and array-like objects
- Develop abstract data types in the most effective way, to convert JavaScript into a more flexible and powerful programming language
- Glance into the pros and cons of JavaScript by implementing real-time code examples
- Develop real-time mini-projects by using JavaScript on the server side to develop desktop and mobile applications
- Work on parallel tasks with asynchronous JavaScript
- Improve code maintainability and readability, and boost app performance through JavaScript

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Diving into the JavaScript Core'
- A synopsis of the book’s content
- More information on JavaScript Unlocked
Dmitry Sheiko is a passionate blogger and the author of *Instant Testing with QUnit*.

Dmitry got hooked to computer programming in the late ‘80s. For the last 18 years, he has been in web development. His very first open source contribution was an XSLT-based CMS in 2004. Since then, he has been contributing quite a lot to FOSS. You can find Dmitry's latest works at https://github.com/dsheiko. Currently, he is working as a web developer in the lovely city of Frankfurt am Main at Crytek GmbH.
JavaScript was born as a scripting language at the most inappropriate time—the time of browser wars. It was neglected and misunderstood for a decade and endured six editions. And look at it now! JavaScript has become a mainstream programming language. It has advanced literally everywhere: in large-scale client-side development, server scripting, desktop applications, native mobile programming, game development, DB querying, hardware control, and OS automating. JavaScript acquired a number of subsets such as Objective-J, CoffeeScript, TypeScript, and others. JavaScript is marvelously concise and an expressive language. It features prototype-based OOP, object composition and inheritance, variadic functions, event-driven programming, and non-blocking I/O. However, to exploit the true power of JavaScript, we need to have a deep understanding of language quirks. Moreover, while developing in JavaScript, we will become aware of its numerous pitfalls, and we will need a few tricks to avoid them. The project formerly known as EcmaScript Harmony, was just recently finalized in the specification named EcmaScript 2015, which is more often referred to as ES6. This not only brought the language to the next level, but also introduced a number of new technologies that require attention.

This book aims to guide the reader in understanding the upcoming and existing features of JavaScript. It is fully packed with code recipes that address common programming tasks. The tasks are supplied with solutions for classical JavaScript (ES5) as well as for the next generation language (ES6-7). The book doesn't focus only on in-browser language, but also provides the essentials on writing efficient JavaScript for desktop applications, server-side software, and native module apps. The ultimate goal of the author is not just to describe the language, but also to help the reader to improve their code for better maintainability, readability, and performance.
What this book covers

Chapter 1, Diving into the JavaScript Core, discusses the techniques to improve the expressiveness of the code, to master multi-line strings and templating, and to manipulate arrays and array-like objects. The chapter explains how to take advantage of JavaScript prototype without harming the readability your code. Further, the chapter introduces the "magic methods" of JavaScript and gives a practical example of their use.

Chapter 2, Modular Programming with JavaScript, describes the modularity in JavaScript: what modules are, why they are important, the standard approaches for asynchronously and synchronously loaded modules, and what ES6 modules are. The chapter shows how CommonJS modules are used in server-side JavaScript and how to pre-compile them for in-browser use. It elaborates how asynchronous and synchronous approaches can be combined to achieve a better application performance. It also explains how one can polyfill ES6 modules for production by the means of Babel.js.

Chapter 3, DOM Scripting and AJAX, introduces Document Object Model (DOM), shows the best practices to minimize browser reflow, and enhance application performance while operating with the DOM. The chapter also compares two client-server communication models: XHR and Fetch API.

Chapter 4, HTML5 APIs, considers the persistence APIs of the browser such as Web Storage, IndexDB, and FileSystem. It introduces Web Components and gives a walk-through of the creation of a custom component. The chapter describes server-to-browser communication APIs such as SSE and WebSockets.

Chapter 5, Asynchronous JavaScript, explains the nonblocking nature of JavaScript, elaborates the event loop and the call stack. The chapter considers the popular styles of chaining asynchronous calls and handling errors. It presents the async/await technique of ES7 and also gives examples of running tasks in parallel and in sequence using the Promise API and the Async.js library. It describes throttling and debouncing concepts.

Chapter 6, A Large-Scale JavaScript Application Architecture, focuses on code maintainability and architecture. The chapter introduces the MVC paradigm and its derivatives, MVP and MVVM. It also shows, with examples, how concern separation is implemented in popular frameworks such as Backbone.js, AngularJS, and ReactJS.
Chapter 7, *JavaScript Beyond the Browser*, explains how to write command-line programs in JavaScript and how to build a web server with Node.js. It also covers the creation of desktop HTML5 applications with NW.js and guides the development of native mobile applications with Phongap.

Chapter 8, *Debugging and Profiling*, dives into bug detection and isolation. It examines the capacities of DevTools and the lesser-known features of the JavaScript console API.
You may have owned an iPhone for years and regard yourself as an experienced user. At the same time, you keep removing unwanted characters one at a time while typing by pressing delete. However, one day you find out that a quick shake allows you to delete the whole message in one tap. Then you wonder why on earth you didn't know this earlier. The same thing happens with programming. We can be quite satisfied with our coding until, all of sudden, we run into a trick or a lesser-known language feature that makes us reconsider the entire work done over the years. It turns out that we could do this in a cleaner, more readable, more testable, and more maintainable way. So it's presumed that you already have experience with JavaScript; however, this chapter equips you with the best practices to improve your code. We will cover the following topics:

- Making your code readable and expressive
- Mastering multiline strings in JavaScript
- Manipulating arrays in the ES5 way
- Traversing an object in an elegant, reliable, safe, and fast way
- The most effective way of declaring objects
- How to magic methods in JavaScript
Make your code readable and expressive
There are numerous practices and heuristics to make a code more readable, expressive, and clean. We will cover this topic later on, but here we will talk about syntactic sugar. The term means an alternative syntax that makes the code more expressive and readable. In fact, we already had some of this in JavaScript from the very beginning. For instance, the increment/decrement and addition/subtraction assignment operators inherited from C. foo++ is syntactic sugar for foo = foo + 1, and foo += bar is a shorter form for foo = foo + bar. Besides, we have a few tricks that serve the same purpose.

JavaScript applies logical expressions to so-called short-circuit evaluation. This means that an expression is read left to right, but as soon as the condition result is determined at an early stage, the expression tail is not evaluated. If we have true || false || false, the interpreter will know from the first test that the result is true regardless of other tests. So the false || false part is not evaluated, and this opens a way for creativity.

Function argument default value
When we need to specify default values for parameters we can do like that:

```javascript
function stub( foo ) {
  return foo || "Default value";
}

console.log( stub( "My value" ) ); // My value
console.log( stub() ); // Default value
```

What is going on here? When foo is true (not undefined, NaN, null, false, 0, or ""), the result of the logical expression is foo otherwise the expression is evaluated until Default value and this is the final result.

Starting with 6th edition of EcmaScript (specification of JavaScript language) we can use nicer syntax:

```javascript
function stub( foo = "Default value" ) {
  return foo;
}
```
Conditional invocation

While composing our code we shorten it on conditions:

```javascript
var age = 20;
age >= 18 && console.log( "You are allowed to play this game" );
age >= 18 || console.log( "The game is restricted to 18 and over" );
```

In the preceding example, we used the AND (`&&`) operator to invoke `console.log` if the left-hand condition is Truthy. The OR (`||`) operator does the opposite, it calls `console.log` if the condition is Falsy.

I think the most common case in practice is the shorthand condition where the function is called only when it is provided:

```javascript
/**
 * @param {Function} [cb] - callback
 */
function fn( cb ) {
    cb && cb();
};
```

The following is one more example on this:

```javascript
/**
 * @class AbstractFoo
 */
AbstractFoo = function(){
    // call this.init if the subclass has init method
    this.init && this.init();
};
```

Syntactic sugar was introduced to its full extent to the JavaScript world only with the advance in CoffeeScript, a subset of the language that trans-compiles (compiles source-to-source) into JavaScript. Actually CoffeeScript, inspired by Ruby, Python, and Haskell, has unlocked arrow-functions, spreads, and other syntax to JavaScript developers. In 2011, Brendan Eich (the author of JavaScript) admitted that CoffeeScript influenced him in his work on EcmaScript Harmony, which was finalized this summer in ECMA-262 6th edition specification. From a marketing perspective, the specification writers agreed on using a new name convention that calls the 6th edition as EcmaScript 2015 and the 7th edition as EcmaScript 2016. Yet the community is used to abbreviations such as ES6 and ES7. To avoid confusion further in the book, we will refer to the specifications by these names. Now we can look at how this affects the new JavaScript.
Diving into the JavaScript Core

Arrow functions

Traditional function expression may look like this:

```javascript
function( param1, param2 ){ /* function body */ }
```

When declaring an expression using the arrow function (aka fat arrow function) syntax, we will have this in a less verbose form, as shown in the following:

```javascript
( param1, param2 ) => { /* function body */ }
```

In my opinion, we don't gain much with this. But if we need, let's say, an array method callback, the traditional form would be as follows:

```javascript
function( param1, param2 ){ return expression; }
```

Now the equivalent arrow function becomes shorter, as shown here:

```javascript
( param1, param2 ) => expression
```

We may do filtering in an array this way:

```javascript
// filter all the array elements greater than 2
var res = [ 1, 2, 3, 4 ].filter(function( v ){ return v > 2; });
console.log( res ); // [3,4]
```

Using an array function, we can do filtering in a cleaner form:

```javascript
var res = [ 1, 2, 3, 4 ].filter( v => v > 2 );
console.log( res ); // [3,4]
```

Besides shorter function declaration syntax, the arrow functions bring the so-called lexical `this`. Instead of creating its own context, it uses the context of the surrounding object as shown here:

```javascript
"use strict";
/**
* @class View
*/
let View = function(){
  let button = document.querySelector( "[data-bind="btn"]" );
  /**
  * Handle button clicked event
  * @private
  */
}
this.onClick = function(){
  console.log( "Button clicked" );
};
button.addEventListener( "click", () => {
  // we can safely refer surrounding object members
  this.onClick();
}, false );
}

In the preceding example, we subscribed a handler function to a DOM event (click). Within the scope of the handler, we still have access to the view context (this), so we don’t need to bind the handler to the outer scope or pass it as a variable through the closure:

var that = this;
button.addEventListener( "click", function(){
  // cross-cutting concerns
  that.onClick();
}, false );

Method definitions

As mentioned in the preceding section, arrow functions can be quite handy when declaring small inline callbacks, but always applying it for a shorter syntax is controversial. However, ES6 provides new alternative method definition syntax besides the arrow functions. The old-school method declaration may look as follows:

var foo = {
  bar: function( param1, param2 ) {
  }
}

In ES6 we can get rid of the function keyword and the colon. So the preceding code can be put this way:

let foo = {
  bar ( param1, param2 ) {
  }
}
The rest operator
Another syntax structure that was borrowed from CoffeeScript came to JavaScript as
the rest operator (albeit, the approach is called splats in CoffeeScript).

When we had a few mandatory function parameters and an unknown number of rest
parameters, we used to do something like this:

```
"use strict";
var cb = function() {
    // all available parameters into an array
    var args = [].slice.call( arguments ),
        // the first array element to foo and shift
        foo = args.shift(),
        // the new first array element to bar and shift
        bar = args.shift();
    console.log( foo, bar, args );
};
cb( "foo", "bar", 1, 2, 3 ); // foo bar [1, 2, 3]
```

Now check out how expressive this code becomes in ES6:

```
let cb = function( foo, bar, ...args ) {
    console.log( foo, bar, args );
}
cb( "foo", "bar", 1, 2, 3 ); // foo bar [1, 2, 3]
```

Function parameters aren't the only application of the rest operator. For example,
we can use it in destructions as well, as follows:

```
let [ bar, ...others ] = [ "bar", "foo", "baz", "qux" ];
console.log( [ bar, others ]); // ["bar","foo","baz","qux"]
```

The spread operator
Similarly, we can spread array elements into arguments:

```
let args = [ 2015, 6, 17 ],
    relDate = new Date( ...args );
console.log( relDate.toString() ); // Fri Jul 17 2015 00:00:00
    GMT+0200 (CEST)
```

ES6 also provides expressive syntactic sugar for object creation and inheritance, but
we will examine this later in The most effective way of declaring objects section.
Mastering multiline strings in JavaScript

Multi-line strings aren’t a good part of JavaScript. While they are easy to declare in other languages (for instance, NOWDOC), you cannot just keep single-quoted or double-quoted strings in multiple lines. This will lead to syntax error as every line in JavaScript is considered as a possible command. You can set backslashes to show your intention:

```
var str = "Lorem ipsum dolor sit amet, \n\nconsectetur adipiscing elit. Nunc ornare, \n\ndiam ultricies vehicula aliquam, mauris \n\nipsum dapibus dolor, quis fringilla leo ligula non neque";
```

This kind of works. However, as soon as you miss a trailing space, you get a syntax error, which is not easy to spot. While most script agents support this syntax, it’s, however, not a part of the EcmaScript specification.

In the times of EcmaScript for XML (E4X), we could assign a pure XML to a string, which opened a way for declarations such as these:

```
var str = <>Lorem ipsum dolor sit amet, 
consectetur adipiscing elit. Nunc ornare, 
diam ultricies vehicula aliquam, mauris 
ipsum dapibus dolor, quis fringilla leo ligula non neque</>.toString();
```

Nowadays E4X is deprecated, it’s not supported anymore.

Concatenation versus array join

We can also use string concatenation. It may feel clumsy, but it's safe:

```
var str = "Lorem ipsum dolor sit amet, \nconsectetur adipiscing elit. Nunc ornare,\n\ndiam ultricies vehicula aliquam, mauris \n\nipsum dapibus dolor, quis fringilla leo ligula non neque";
```

You may be surprised, but concatenation is slower than array joining. So the following technique will work faster:

```
var str = [ "Lorem ipsum dolor sit amet, \n\nconsectetur adipiscing elit. Nunc ornare,\n\ndiam ultricies vehicula aliquam, mauris \n\n\ndiam ultricies vehicula aliquam, mauris \n\n\nipsum dapibus dolor, quis fringilla leo ligula non neque"].join( "\n\" );
```
Template literal

What about ES6? The latest EcmaScript specification introduces a new sort of string literal, template literal:

```javascript
var str = `Lorem ipsum dolor sit amet, 
consectetur adipiscing elit. Nunc ornare, 
diam ultricies vehicula aliquam, mauris
ipsum dapibus dolor, quis fringilla leo ligula non neque`;
```

Now the syntax looks elegant. But there is more. Template literals really remind us of NOWDOC. You can refer any variable declared in the scope within the string:

```javascript
"use strict";
var title = "Some title",
    text = "Some text",
    str = `<div class="message">
    <h2>${title}</h2>
    <article>${text}</article>
</div>`;
console.log( str );
```

The output is as follows:

```html
<div class="message">
    <h2>Some title</h2>
    <article>Some text</article>
</div>
```

If you wonder when can you safely use this syntax, I have a good news for you—this feature is already supported by (almost) all the major script agents (http://kangax.github.io/compat-table/es6/).

Multi-line strings via transpilers

With the advance of ReactJS, Facebook's EcmaScript language extension named JSX (https://facebook.github.io/jsx/) is now really gaining momentum. Apparently influenced by previously mentioned E4X, they proposed a kind of string literal for XML-like content without any screening at all. This type supports template interpolation similar to ES6 templates:

```javascript
"use strict";
var Hello = React.createClass({
    render: function() {
        return <div class="message">
            <h2>{this.props.title}</h2>
        </div>;
    }
});
```
Another way to declare multiline strings is by using CommonJS Compiler (http://dsheiko.github.io/cjsc/). While resolving the 'require' dependencies, the compiler transforms any content that is not .js/.json content into a single-line string:

```javascript
var str = require( "./foo.txt" );
console.log( str );
```

You can find an example of JSX use in Chapter 6, A Large-Scale JavaScript Application Architecture.

**Manipulating arrays in the ES5 way**

Some years ago when the support of ES5 features was poor (EcmaScript 5th edition was finalized in 2009), libraries such as Underscore and Lo-Dash got highly popular as they provided a comprehensive set of utilities to deal with arrays/collections. Today, many developers still use third-party libraries (including jQuery/Zepro) for methods such as map, filter, every, some, reduce, and indexOf, while these are available in the native form of JavaScript. It still depends on how you use such libraries, but it may likely happen that you don't need them anymore. Let's see what we have now in JavaScript.
Array methods in ES5

Array.prototype.forEach is probably the most used method of the arrays. That is, it is the native implementation of __.each, or for example, of the $.each utilities. As parameters, forEach expects an iteratee callback function and optionally a context in which you want to execute the callback. It passes to the callback function an element value, an index, and the entire array. The same parameter syntax is used for most array manipulation methods. Note that jQuery's $.each has the inverted callback parameters order:

```
"use strict";
var data = [ "bar", "foo", "baz", "qux" ];
data.forEach(function( val, inx ){
    console.log( val, inx );
});
```

Array.prototype.map produces a new array by transforming the elements of a given array:

```
"use strict";
var data = { bar: "bar bar", foo: "foo foo" },
    // convert key-value array into url-encoded string
urlEncStr = Object.keys( data ).map(function( key ){
    return key + "=" + window.encodeURIComponent( data[ key ] );
}).join( ";" );

console.log( urlEncStr ); // bar=bar%20bar&foo=foo%20foo
```

Array.prototype.filter returns an array, which consists of given array values that meet the callback's condition:

```
"use strict";
var data = [ "bar", "foo", "", 0 ],
    // remove all falsy elements
filtered = data.filter(function( item ){ return !!item; });

console.log( filtered ); // ["bar", "foo"]
```

Array.prototype.reduce/Array.prototype.reduceRight retrieves the product of values in an array. The method expects a callback function and optionally the initial value as arguments. The callback function receive four parameters: the accumulative value, current one, index and original array. So we can, for an instance, increment the accumulative value by the current one (return acc += cur;) and, thus, we will get the sum of array values.
Besides calculating with these methods, we can concatenate string values or arrays:

```
"use strict";
var data = [[0, 1], [2, 3], [4, 5]],
    arr = data.reduce(function(prev, cur) {
        return prev.concat(cur);
    }),
    arrReverse = data.reduceRight(function(prev, cur) {
        return prev.concat(cur);
    });

console.log(arr); // [0, 1, 2, 3, 4, 5]
console.log(arrReverse); // [4, 5, 2, 3, 0, 1]
```

Array.prototype.some tests whether any (or some) values of a given array meet the callback condition:

```
"use strict";
var bar = ["bar", "baz", "qux"],
    foo = ["foo", "baz", "qux"],
    /**
    * Check if a given context (this) contains the value
    * @param {*} val
    * @return {Boolean}
    */
    compare = function(val){
        return this.indexOf(val) !== -1;
    },

console.log(bar.some(compare, foo)); // true
```

In this example, we checked whether any of the bar array values are available in the foo array. For testability, we need to pass a reference of the foo array into the callback. Here we inject it as context. If we need to pass more references, we would push them in a key-value object.

As you probably noticed, we used in this example Array.prototype.indexOf. The method works the same as String.prototype.indexOf. This returns an index of the match found or -1.

Array.prototype.every tests whether every value of a given array meets the callback condition:

```
"use strict";
var bar = ["bar", "baz"],
    foo = ["bar", "baz", "qux"],
```
Diving into the JavaScript Core

/**
 * Check if a given context (this) contains the value
 * @param {*} val
 * @return {Boolean}
 */
compare = function( val ){
    return this.indexOf( val ) !== -1;
};

console.log( bar.every( compare, foo ) ); // true

If you are still concerned about support for these methods in a legacy browser as old as IE6-7, you can simply shim them with https://github.com/es-shims/es5-shim.

Array methods in ES6
In ES6, we get just a few new methods that look rather like shortcuts over the existing functionality.

Array.prototype.fill populates an array with a given value, as follows:

    "use strict";
    var data = Array( 5 );
    console.log( data.fill( "bar" ) ); // ["bar", "bar", "bar", "bar", "bar"]

Array.prototype.includes explicitly checks whether a given value exists in the array. Well, it is the same as arr.indexOf( val ) !== -1, as shown here:

    "use strict";
    var data = [ "bar", "foo", "baz", "qux" ];
    console.log( data.includes( "foo" ) );

Array.prototype.find filters out a single value matching the callback condition. Again, it's what we can get with Array.prototype.filter. The only difference is that the filter method returns either an array or a null value. In this case, this returns a single element array, as follows:

    "use strict";
    var data = [ "bar", "fo", "baz", "qux" ],
    match = function( val ){
        return val.length < 3;
    };
    console.log( data.find( match ) ); // fo
Traversing an object in an elegant, reliable, safe, and fast way

It is a common case when we have a key-value object (let's say options) and need to iterate it. There is an academic way to do this, as shown in the following code:

```
"use strict";
var options = {
    bar: "bar",
    foo: "foo"
},
key;
for( key in options ) {
    console.log( key, options[ key] );
}
```

The preceding code outputs the following:

```
bar bar
foo foo
```

Now let's imagine that any of the third-party libraries that you load in the document augments the built-in `Object`:

```
Object.prototype.baz = "baz";
```

Now when we run our example code, we will get an extra undesired entry:

```
bar bar
foo foo
baz baz
```

The solution to this problem is well known, we have to test the keys with the `Object.prototype.hasOwnProperty` method:

```
//...
for( key in options ) {
    if ( options.hasOwnProperty( key ) ) {
        console.log( key, options[ key] );
    }
}
```
Iterating the key-value object safely and fast

Let’s face the truth—the structure is clumsy and requires optimization (we have to perform the `hasOwnProperty` test on every given key). Luckily, JavaScript has the `Object.keys` method that retrieves all string-valued keys of all enumerable own (non-inherited) properties. This gives us the desired keys as an array that we can iterate, for instance, with `Array.prototype.forEach`:

```javascript
"use strict";
var options = {
  bar: "bar",
  foo: "foo"
};
Object.keys( options ).forEach(function( key ){
  console.log( key, options[ key] );
});
```

Besides the elegance, we get a better performance this way. In order to see how much we gain, you can run this online test in distinct browsers such as: `http://codepen.io/dsheiko/pen/JdrqXa`.

Enumerating an array-like object

Objects such as `arguments` and `nodeList` (from `node.querySelectorAll`, `document.forms`) look like arrays, in fact they are not. Similar to arrays, they have the `length` property and can be iterated in the `for` loop. In the form of objects, they can be traversed in the same way that we previously examined. But they do not have any of the array manipulation methods (`forEach`, `map`, `filter`, `some` and so on). The thing is we can easily convert them into arrays as shown here:

```javascript
"use strict";
var nodes = document.querySelectorAll( "div" ),
  arr = Array.prototype.slice.call( nodes );

arr.forEach(function(i){
  console.log(i);
});
```

The preceding code can be even shorter:

```javascript
arr = [].slice.call( nodes )
```

It’s a pretty convenient solution, but looks like a trick. In ES6, we can do the same conversion with a dedicated method:

```javascript
arr = Array.from( nodes );
```
The collections of ES6

ES6 introduces a new type of objects—iterable objects. These are the objects whose elements can be retrieved one at a time. They are quite the same as iterators in other languages. Beside arrays, JavaScript received two new iterable data structures, `Set` and `Map`. `Set` which are a collection of unique values:

```
"use strict";
let foo = new Set();
foo.add( 1 );
foo.add( 1 );
foo.add( 2 );
console.log( Array.from( foo ) ); // [ 1, 2 ]
```

```
let foo = new Set(),
    bar = function(){ return "bar"; };
foo.add( bar );
console.log( foo.has( bar ) ); // true
```

The map is similar to a key-value object, but may have arbitrary values for the keys. And this makes a difference. Imagine that we need to write an element wrapper that provides jQuery-like events API. By using the `on` method, we can pass not only a handler callback function but also a context (`this`). We bind the given callback to the `cb.bind( context ) context. This means `addEventListener` receives a function reference different from the callback. How do we unsubscribe the handler then? We can store the new reference in `Map` by a key composed from an event name and a callback function reference:

```
"use strict";
/**
 * @class
 * @param {Node} el
 */
let El = function( el ){
    this.el = el;
    this.map = new Map();
};
/**
 * Subscribe a handler on event
 * @param {String} event
 * @param {Function} cb
 * @param {Object} context
 */
El.prototype.on = function( event, cb, context ){
    let handler = cb.bind( context || this );
```
this.map.set( [ event, cb ], handler );
this.el.addEventListener( event, handler, false );
};
/**
 * Unsubscribe a handler on event
 * @param {String} event
 * @param {Function} cb
 */
El.prototype.off = function( event, cb ){
  let handler = cb.bind( context ),
      key = [ event, handler ];
  if ( this.map.has( key ) ) {
    this.el.removeEventListener( event, this.map.get( key ) );
    this.map.delete( key );
  }
};

Any iterable object has methods, keys, values, and entries, where the keys work
the same as Object.keys and the others return array values and an array of key-
value pairs respectively. Now let's see how we can traverse the iterable objects:

"use strict";
let map = new Map()
  .set( "bar", "bar" )
  .set( "foo", "foo" ),
  pair;
for ( pair of map ) {
  console.log( pair );
}

// OR
let map = new Map([
  [ "bar", "bar" ],
  [ "foo", "foo" ],
]);
map.forEach(function( value, key ){
  console.log( key, value );
});

Iterable objects have manipulation methods such as arrays. So we can use forEach.
Besides, they can be iterated by for...in and for...of loops. The first one retrieves
indexes and the second, the values.
The most effective way of declaring objects
How do we declare an object in JavaScript? If we need a namespace, we can simply use an object literal. But when we need an object type, we need to think twice about what approach to take, as it affects the maintainability of our object-oriented code.

Classical approach
We can create a constructor function and chain the members to its context:

```javascript
"use strict";
/**
 * @class
 */
var Constructor = function(){
/**
 * @type {String}
 * @public
 */
this.bar = "bar";
/**
 * @public
 * @returns {String}
 */
this.foo = function() {
    return this.bar;
};
},
/** @type Constructor */
instance = new Constructor();

console.log( instance.foo() ); // bar
console.log( instance instanceof Constructor ); // true
```

We can also assign the members to the constructor prototype. The result will be the same as follows:

```javascript
"use strict";
/**
 * @class
 */
var Constructor = function(){},
instance;
```
Diving into the JavaScript Core

```javascript
/**
 * @type {String}
 * @public
 */
Constructor.prototype.bar = "bar";
/**
 * @public
 * @returns {String}
 */
Constructor.prototype.foo = function() {
    return this.bar;
};
/** @type Constructor */
instance = new Constructor();

console.log( instance.foo() ); // bar
console.log( instance instanceof Constructor ); // true
```

In the first case, we have the object structure within the constructor function body mixed with the construction logic. In the second case by repeating `Constructor.prototype`, we violate the Do Not Repeat Yourself (DRY) principle.

**Approach with the private state**

So how can we do it otherwise? We can return an object literal by the constructor function:

```
"use strict";
/**
 * @class
 */
var Constructor = function(){
    /**
     * @type {String}
     * @private
     */
    var baz = "baz";
    return {
        /**
         * @type {String}
         * @public
         */
        bar: "bar",
        /**
         */
    }
```
bam: function() {
    return this.bar + " "+ baz;
}
});
/** @type Constructor */
instance = new Constructor();

console.log( instance.foo() ); // bar baz
console.log( instance.hasOwnProperty("baz") ); // false
console.log( Constructor.prototype.hasOwnProperty("baz") ); // false
console.log( instance instanceof Constructor ); // false

The advantage of this approach is that any variables declared in the scope of the
constructor are in the same closure as the returned object, and therefore, available
through the object. We can consider such variables as private members. The bad
news is that we will lose the constructor prototype. When a constructor returns an
object during instantiation, this object becomes the result of a whole new expression.

## Inheritance with the prototype chain

What about inheritance? The classical approach would be to make the subtype
prototype an instance of supertype:

```
"use strict";
/**
 * @class
 */
var SuperType = function(){
    /**
     * @type {String}
     * @public
     */
    this.foo = "foo";
},
/**
 * @class
 */
Constructor = function(){
    /**
     * @type {String}
     */
```

[19]
Diving into the JavaScript Core

```javascript
/* @public */
this.bar = "bar";

/** @type Constructor */
instance;

Constructor.prototype = new SuperType();
Constructor.prototype.constructor = Constructor;

instance = new Constructor();
console.log( instance.bar ); // bar
console.log( instance.foo ); // foo
console.log( instance instanceof Constructor ); // true
console.log( instance instanceof SuperType ); // true

You may run into some code, where for instantiation `Object.create` is used instead of the new operator. Here you have to know the difference between the two. `Object.create` takes an object as an argument and creates a new one with the passed object as a prototype. In some ways, this reminds us of cloning. Examine this, you declare an object literal (proto) and create a new object (instance) with `Object.create` based on the first one. Whatever changes you do now on the newly created object, they won't be reflected on the original (proto). But if you change a property of the original, you will find the property changed in its derivative (instance):

"use strict";
var proto = {
  bar: "bar",
  foo: "foo"
},
instance = Object.create( proto );
proto.bar = "qux",
instance.foo = "baz";
console.log( instance ); // { foo="baz", bar="qux"}
console.log( proto ); // { bar="qux", foo="foo"}

Inheriting from prototype with `Object.create`

In contrast to the new operator, `Object.create` does not invoke the constructor. So when we use it to populate a subtype prototype, we are losing all the logic located in a supertype constructor. This way, the supertype constructor is never called:

// ...
SuperType.prototype.baz = "baz";
Constructor.prototype = Object.create( SuperType.prototype );
```
Constructor.prototype.constructor = Constructor;

instance = new Constructor();

console.log( instance.bar ); // bar
console.log( instance.baz ); // baz
console.log( instance.hasOwnProperty( "foo" ) ); // false
console.log( instance instanceof Constructor ); // true
console.log( instance instanceof SuperType ); // true

Inheriting from prototype with Object.assign

When looking for an optimal structure, I would like to declare members via an object literal, but still have the link to the prototype. Many third-party projects leverage a custom function (extend) that merge the structure object literal into the constructor prototype. Actually, ES6 provides an Object.assign native method.

We can use it as follows:

"use strict";
/**
 * @class
 */
var SuperType = function(){
 /**
 * @type {String}
 * @public
 */
 this.foo = "foo";
},
/**
 * @class
 */
Constructor = function(){
 /**
 * @type {String}
 * @public
 */
 this.bar = "bar";
},
/** @type Constructor */
instance;

Object.assign( Constructor.prototype = new SuperType(), { baz: "baz"
Diving into the JavaScript Core

```javascript
instance = new Constructor();
console.log( instance.bar ); // bar
console.log( instance.foo ); // foo
console.log( instance.baz ); // baz
console.log( instance instanceof Constructor ); // true
console.log( instance instanceof SuperType ); // true
```

This looks almost as required, except there is one inconvenience. `Object.assign` simply assigns the values of source objects to the target ones regardless of their type. So if you have a source property with an object (for instance, an `Object` or `Array` instance), the target object receives a reference instead of a value. So you have to reset manually any object properties during initialization.

## Approach with ExtendClass

ExtendClass, proposed by Simon Boudrias, is a solution that seems flawless (https://github.com/SBoudrias/class-extend). His little library exposes the `Base` constructor with the `extend` static method. We use this method to extend this pseudo-class and any of its derivatives:

```javascript
"use strict";
/**
 * @class
 * /
 var SuperType = Base.extend(
 /**
 * @public
 * @returns {String}
 * /
 foo: function(){ return "foo public"; },
 /**
 * @constructs SuperType
 * /
 constructor: function () {}
 ),
 /**
 * @class
 * /
 Constructor = SuperType.extend(
 /**
 * @public
 * @returns {String}
 * /
```
bar: function(){ return "bar public"; }
}, {
/**
 * @static
 * @returns {String}
 */
bar: function(){ return "bar static"; }
}),
/** @type Constructor */
instance = new Constructor();

console.log( instance.foo() ); // foo public
console.log( instance.bar() ); // bar public
console.log( Constructor.bar() ); // bar static
console.log( instance instanceof Constructor ); // true
console.log( instance instanceof SuperType ); // true

Classes in ES6
TC39 (the EcmaScript working group) is pretty aware of the problem, so the new language specification provides extra syntax to structure object types:

"use strict";
class AbstractClass {
constructor() {
    this.foo = "foo";
}
}
class ConcreteClass extends AbstractClass {
constructor() {
    super();
    this.bar = "bar";
}
baz() {
    return "baz";
}
}

let instance = new ConcreteClass();
console.log( instance.bar ); // bar
console.log( instance.foo ); // foo
console.log( instance.baz() ); // baz
console.log( instance instanceof ConcreteClass ); // true
console.log( instance instanceof AbstractClass ); // true
The syntax looks class-based, but in fact this a syntactic sugar over existing prototypes. You can check with the type of `ConcreteClass`, and it will give you `function` because `ConcreteClass` is a canonical constructor. So we don't need any trick to extend supertypes, no trick to refer the supertype constructor from subtype, and we have a clean readable structure. However, we cannot assign properties the same C-like way as we do now with methods. This is still in discussion for ES7 (https://esdiscuss.org/topic/es7-property-initializers). Besides this, we can declare a class's static methods straight in its body:

```javascript
class Bar {
  static foo() {
    return "static method";
  }
  baz() {
    return "prototype method";
  }
}
let instance = new Bar();
console.log( instance.baz() ); // prototype method
console.log( Bar.foo() ); // static method
```

Actually, there are many in the JavaScript community who consider the new syntax as a deviation from the prototypical OOP approach. On the other hand, the ES6 classes are backwards compatible with most of the existing code. Subclasses are now supported by the language and no extra libraries are required for inheritance. And what I personally like the most is that this syntax allows us to make the code cleaner and more maintainable.

### How to – magic methods in JavaScript

In the PHP world, there are things such as overloading methods, which are also known as magic methods (http://www.php.net/manual/en/language.oop5.overloading.php). These methods allow us to set a logic that triggers when a nonexistent property of a method is being accessed or modified. In JavaScript, we control access to properties (value members). Imagine we have a custom collection object. In order to be consistent in the API, we want to have the `length` property that contains the size of the collection. So we declare a getter (get length), which does the required computation whenever the property is accessed. On attempting to modify the property value, the setter will throw an exception:

```javascript
"use strict";
var bar = {
  /** @type {([Number])} */
  arr: [ 1, 2 ],
```
Chapter 1

/**
 * Getter
 * @returns {Number}
*/
get length () {
    return this.arr.length;
},
/**
 * Setter
 * @param {*} val
 */
set length ( val ) {
    throw new SyntaxError( "Cannot assign to read only property 'length'" );
}
); console.log ( bar.length ); // 2
bar.arr.push( 3 );
console.log ( bar.length ); // 3
bar.length = 10; // SyntaxError: Cannot assign to read only property 'length'

If we want to declare getters/setters on an existing object, we can use the following:

Object.defineProperty:
"use strict";
var bar = {
    /** @type {[Number]} */
    arr: [ 1, 2 ]
};

Object.defineProperty( bar, "length", {
    /**
     * Getter
     * @returns {Number}
     */
    get: function() {
        return this.arr.length;
    },
    /**
     * Setter
     */
    set: function() {
        throw new SyntaxError( "Cannot assign to read only property 'length'" );
    }
});
Object.defineProperty as well as the second parameter of Object.create specifies a property configuration (whether it is enumerable, configurable, immutable, and how it can be accessed or modified). So, we can achieve a similar effect by configuring the property as read-only:

```javascript
"use strict";
var bar = {};

Object.defineProperty( bar, "length", {
    /**
     * Data descriptor
     * @type {*} 
     */
    value: 0,
    /**
     * Data descriptor
     * @type {Boolean}
     */
    writable: false
});

bar.length = 10; // TypeError: "length" is read-only
```

By the way, if you want to get rid of the property accessor in the object, you can simply remove the property:

```javascript
delete bar.length;
```

## Accessors in ES6 classes

Another way by which we can declare accessors is using the ES6 classes:

```javascript
"use strict";
/** @class */
class Bar {
    /** @constructs Bar */
    constructor() {
```
/** @type {([Number])} */
this.arr = [ 1, 2 ];
}
/**
 * Getter
 * @returns {Number}
*/
get length() {
  return this.arr.length;
}
/**
 * Setter
 * @param {Number} val
 */
set length( val ) {
  throw new SyntaxError( "Cannot assign to read only property 'length'" );
}

let bar = new Bar();
console.log ( bar.length ); // 2
bar.arr.push( 3 );
console.log ( bar.length ); // 3
bar.length = 10; // SyntaxError: Cannot assign to read only property 'length'

 Besides public properties, we can control access to static ones as well:

"use strict";

class Bar {
  /**
   * @static
   * @returns {String}
   */
  static get baz() {
    return "baz";
  }
}

console.log( Bar.baz ); // baz
Controlling access to arbitrary properties

All these examples show access control to known properties. However, there might be a case when I want a custom storage with a variadic interface similar to `localStorage`. This must be a storage that has the `getItem` method to retrieve stored values and the `setItem` method to set them. Besides, this must work the same way as when you directly access or set a pseudo-property (\(\text{val} = \text{storage.aKey}\) and \(\text{storage.aKey} = \text{"value"}\)). These can be achieved by using the ES6 Proxy:

```javascript
"use strict";
/**
 * Custom storage
 */
var myStorage = {
    /** @type {Object} key-value object */
    data: {},
    /**
     * Getter
     * @param {String} key
     * @returns {*
     */
    getItem: function( key ){
        return this.data[ key ];
    },
    /**
     * Setter
     * @param {String} key
     * @param {*} val
     */
    setItem: function( key, val ){
        this.data[ key ] = val;
    }
},
/**
 * Storage proxy
 * @type {Proxy}
 */
storage = new Proxy( myStorage, {
    /**
     * Proxy getter
     * @param {myStorage} storage
     * @param {String} key
     * @returns {*
     */
```
get: function ( storage, key ) {
    return storage.getItem( key );
},
/**
 * Proxy setter
 * @param {myStorage} storage
 * @param {String} key
 * @param {*} val
 * @returns {void}
 */
set: function ( storage, key, val ) {
    return storage.setItem( key, val );
});

storage.bar = "bar";
console.log( myStorage.getItem( "bar" ) ); // bar
myStorage.setItem( "bar", "baz" );
console.log( storage.bar ); // baz

Summary
This chapter gives practices and tricks on how to use the JavaScript core features for the maximum effect. In the next chapter, we will talk about module concepts and we will do a walkthrough on scopes and closures. The next chapter will explain the scope context and the ways to manipulate it.
Where to buy this book
You can buy JavaScript Unlocked from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.