1

Diving into JavaScript core

You may have an iPhone for years and regard yourself as an experienced user. At the same time you keep removing unwanted characters during typing by pressing delete, one at a time. However one day you find out that a quick shake allows you delete the whole message in one tap. Then you wonder why on earth you didn’t know it earlier. The same 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 of years. It turns out that we could do it cleaner, more readable, testable, and maintainable. So it’s presumed that you already have experience with JavaScript, but the chapter equips you with best practices to improve your code.

Make your code readable and expressive

Mastering multi-line strings in JavaScript

Manipulating arrays ES5 way

Traversing an object: elegant, reliable, safe and fast

Most effective way of declaring objects

How to: magic methods in JavaScript

# Make your code readable and expressive

There quite numerous practices and heuristics as to keep code more readable, expressive and clean. We touch upon that topic later on, but here we talk about syntactic sugar. The term means an alternative syntax that is meant to make the code more expressive and readable. In fact we have some of it in JavaScript from the very beginning, for an instance 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 on logical expressions so called short-circuit evaluation. That mean an expression is read left to right, but as soon as the condition result is determined on early stage the expression tail is not evaluated. If we have true || false || false the interpreter knows from the first test that the result is true regardless of other tests. So false || false part is not evaluated what opens a way for creativity.

## Function argument default value

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 truthy (not undefined, NaN, null, false, 0, “”) the result of the logical expression is foo, otherwise the expression is evaluated till "Default value" and that makes the result.

## Conditional invocation

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 provided example we use AND (&&) operator to invoke console.log if the left-hand condition is true. OR (||) operator does the opposite, it calls console.log if the condition is false.

I think the most common case in real-world practice is a shorthand condition where the function is called only when it is provided.

/\*\*

\* @param {Function} [cb] - callback

\*/

function fn( cb ) {

 cb && cb();

};

or

/\*\*

\* @class AbstractFoo

\*/

AbstractFoo = function(){

 // call this.init if the subclass has init method

 this.init && this.init();

};

To the full extent syntactic sugar came to JavaScript world only with the advance of CoffeeScript, a subset of the language that transcompiles (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 marketing perspective the spec writers agreed on using a new name convention that makes 6th edition - EcmaScript 2015 and 7th - EcmaScript 2016. Yet the community used to abbreviations ES6 and ES7. To avoid the confusion further in the book we will refer the specifications by these names Now we can look at how this affected new JavaScript.

## Arrow functions

Traditional function expression may look like that:

function( param1, param2 ){ /\* function body \*/ }

When declaring the expression using arrow function (aka fat arrow function) syntax, we get it in a less verbose form:

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

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

function( param1, param2 ){ return expression; }

Now the equivalent arrow function gets shorter

( param1, param2 ) => expression

So if we used filtering an array this way:

// 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]

By using an array function we get it in a cleaner form:

var res  = [ 1, 2, 3, 4 ].filter( v => v > 2 );

console.log( res ); // [3,4]

Besides, shorter function declaration syntax the arrow functions bring so called lexical this. Instead of creating their own context, they use the context of the surrounding object

"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 example above we subscribe a handler function on 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 though the closure:

var that = this;

button.addEventListener( "click", function(){

// cross-cutting concerns

  that.onClick();

}, false );

## Method definitions

As it is shown above arrow functions can be quite handy when declaring small inline callbacks, but always going with it  for shorter syntax is disputable. However, ES6 provides new alternative method definition syntax besides the arrow functions. The old-school method declaration may look so:

var foo = {

 bar: function( param1, param2 ) {

 }

}

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

let foo = {

 bar ( param1, param2 ) {

 }

}

## The rest operator

Another syntax structure borrowed from CoffeeScript came to JavaScript as rest operator (albeit the approach is called “splats” in CoffeeScript).

When we have a number of few mandatory function parameters and unknown number of rest parameters we used to do something like that:

"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 it 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 appliance for the rest operator. For example, we can use it in destruction:

let [ bar, ...others ] = [ "bar", "foo", "baz", "qux" ];

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

## The spread operator

The same way 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 examine that later in topic “Most effective way of declaring objects”

# Mastering multi-line strings in JavaScript

Multi-line strings aren’t a good part of JavaScript. While they are so easy to declare in other languages (e.g. NOWDOC), you cannot just keep single- or double-quoted strings multi-line, it gives a syntax error as of every line in JavaScript is considered as a possible command. You can set backslashes to show your intend:

var str = "Lorem ipsum dolor sit amet, \n\

consectetur adipiscing elit. Nunc ornare, \n\

diam ultricies vehicula aliquam, mauris \n\

ipsum dapibus dolor, quis fringilla leo ligula non neque";

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

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

var str = <>Lorem ipsum dolor sit amet,

consectetur adipiscing

elit. Nunc ornare </>.toString();

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

## Concatenation vs array join

We can also go with string concatenation. It may feel clumsy, but it’s safe:

var str = "Lorem ipsum dolor sit amet, \n" +

 "consectetur adipiscing elit. Nunc ornare,\n" +

 "diam ultricies vehicula aliquam, mauris \n" +

 "ipsum dapibus dolor, quis fringilla leo ligula non neque";

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

var str = [ "Lorem ipsum dolor sit amet, \n",

 "consectetur adipiscing elit. Nunc ornare,\n",

 "diam ultricies vehicula aliquam, mauris \n",

 "ipsum dapibus dolor, quis fringilla leo ligula non neque"].join( "" );

## Template literal

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

var str = `Lorem ipsum dolor sit amet, \n

consectetur adipiscing elit. Nunc ornare, \n

diam ultricies vehicula aliquam, mauris \n

ipsum dapibus dolor, quis fringilla leo ligula non neque`;

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

"use strict";

var title = "Some title",

   text = "Some text",

   str = `<div class="message">

<h2>${title}</h2>

<article>${text}</article>

</div>`;

console.log( str );

Output:

<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 traspilers

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 mentioned previously E4X, they proposed a kind of string literal for XML-like content without any screening at all. The type supports template interpolation similar to ES6 templates:

"use strict";

var Hello = React.createClass({

 render: function() {

   return <div class="message">

<h2>{this.props.title}</h2>

<article>{this.props.text}</article>

</div>;

 }

});

React.render(<Hello title="Some title" text="Some text" />, node);

Another way to declare multiline strings worth of mentioning can be achieved by using CommonJS Compiler (http://dsheiko.github.io/cjsc/). While resolving `require` dependencies any non .js/.json content the compiler transforms into a single-line string:

foo.txt

Lorem ipsum dolor sit amet,

consectetur adipiscing elit. Nunc ornare,

diam ultricies vehicula aliquam, mauris

ipsum dapibus dolor, quis fringilla leo ligula non neque

consumer.js

var str = require( "./foo.txt" );

console.log( str );

# Manipulating arrays ES5 way

Some years ago when support of ES5 features was poor (EcmaScript 5th edition was finalized in 2009) such libraries as Underscore and Lo-Dash got highly popular as they provided a comprehensive set of utilities to deal with arrays/collections. Today many developer still use 3rd party libraries (including jQuery/Zepro) for methods such as  map, filter, every, some, reduce, indexOf while those are available in JavaScript native way. It still depends on how you use such libraries, but 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 array. That is the native implementation of \_.each or for example $.each utilities. forEach expects as parameters iteratee callback function and optionally a context in which you want to execute the callback. It passes to callback function element value, 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 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 arrays, which consists of a given array values that meet 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. These are the only methods whose parameter syntax is different.  The methods parameters - the callback and initial value and callback gets accumulative value, current one, index and original array.

Besides calculating (e.g. sum of all values within an array) with these methods we can concat 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 if any (some) values of a given array meet 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 check if any of bar array values available in foo array. For testability we need to pass reference to foo array into the callback. Here we inject it as context. If we needed  to pass more references, we would push them in a key-value object.

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

Array.prototype.every tests if every value of a given array meets callback condition.

"use strict";

var bar = [ "bar", "baz" ],

   foo = [ "bar", "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.every( compare, foo ) ); // true

If you are still concerned about support for these methods in 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 as shortcuts over existing functionality.

Array.prototype.fill populates an array with a given value

"use strict";

var data = Array( 5 );

console.log( data.fill( "bar" ) ); // ["bar", "bar", "bar", "bar", "bar"]

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

"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 the filter method returns either an array or null. In this case a single element array.

"use strict";

var data = [ "bar", "fo", "baz", "qux" ],

   match = function( val ){

     return val.length < 3;

   };

console.log( data.find( match ) ); // fo

# Traversing an object: elegant, reliable, safe and fast

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 so:

"use strict";

var options = {

    bar: "bar",

    foo: "foo"

   },

   key;

for( key in options ) {

 console.log( key, options[ key] );

}

It outputs:

bar bar

foo foo

Now let’s imagine any of 3rd party libraries you load in the document augments the built-in object `Object`:

Object.prototype.baz = "baz";

Now when we run our example code now, we get extra a undesired entry:

bar bar

foo foo

baz baz

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

//…

for( key in options ) {

 if ( options.hasOwnProperty( key ) ) {

   console.log( key, options[ key] );

 }

}

## Iterating key-value object safe and fast

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

"use strict";

var options = {

    bar: "bar",

    foo: "foo"

   };

Object.keys( options ).forEach(function( key ){

 console.log( key, options[ key] );

});

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

## Enumerating array-like object

Objects such as arguments and  nodeList (node.querySelectorAll, document.forms) look like arrays. In fact they are not. Alike arrays they have `length` property and can be iterated in `for` loop. By being objects they can be traversed the way we examined just before. But they do not have any of array manipulation methods (forEach, map, filter, some and so on). The thing is we can easily convert them into arrays.

"use strict";

var nodes = document.querySelectorAll( "div" ),

   arr = Array.prototype.slice.call( nodes );

arr.forEach(function(i){

 console.log(i);

});

Or even shorter:

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:

arr = Array.from( nodes );

## Collections of ES6

ES6 introduces new types of objects - iterable objects. These are objects whose elements can be retrieved one at a time, quite the same as iterators in other languages. Beside arrays, JavaScript received 2 new iterable data structures Set and Map. Set is 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

Map is similar to a key-value object, but may have arbitrary values for the keys. And that makes a difference. Imagine that we need to write an element wrapper that provides jQuery-like events API. By using `on` method we can pass not only a handler callback function, but also a context (this). We bind the given callback to the context cb.bind( context ). It means addEventListener receives a function reference different from the callback. How do we unsubscribe the handler then? We can store the new reference in a Map by a key composed from event name and 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 keys work the same as Object.keys and other return array values and array of key-value pairs respectively. Now let’s see how can we 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 object have manipulation methods like arrays. So we can go with forEach. Besides, they can be iterated by for...in and for...of loops. The first one retrieves indexes and the second the values.

# Most effective way of declaring objects

How do we declare an object in JavaScript? If we need a namespace we can simply go with object literal. But when we need an object type we need to think twice what approach to take as it affects maintainability of our object-oriented code.

## Classical approach

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

"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:

"use strict";

/\*\*

\* @class

\*/

var Constructor = function(){},

   instance;

/\*\*

\* @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 DRY (Do Not Repeat Yourself)  principle.

## Approach with private state

So how can we do it else? We can return in the constructor function an object literal.

"use strict";

/\*\*

 \* @class

 \*/

var Constructor = function(){

     /\*\*

     \* @type {String}

     \* @private

     \*/

     var baz = "baz";

     return {

       /\*\*

       \* @type {String}

       \* @public

       \*/

       bar: "bar",

       /\*\*

       \* @public

       \* @returns {String}

       \*/

       foo: 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 bonus of this approach is that any variables declared in the scope of the constructor are in the same closure with the returned object and, therefore, available through the object. We can consider such variables as private members. The bad news is that we lost the constructor prototype. When constructor returns an object during instantiation, that object becomes the result of the 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}

       \* @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 new operator. Here you have to know the difference. Object.create takes an object as an argument and creates a new one with the passed in object as a prototype. In some ways it reminds 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 reflect on the origin (proto). But if you change the a property of the origin, you will find it 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 new operator, Object.create does not invoke the constructor. So when we use it to populate subtype prototype we are losing all the logic located in supertype constructor. This way 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 the optimal structure I would like to declare members via object literal, but still have the link to the prototype. Many 3rd party projects leverage a custom function (extend), that merge the structure object literal into the constructor prototype. Actually ES6 provides a native methods Object.assign. We can use it:

"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"

});

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 desired, except one inconvenience. Object.assign simply assigns values of sources objects to the target one regardless of their types. So if you have a source property with an object (e.g. Object, 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

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

"use strict";

   /\*\*

    \* @class

    \*/

var SuperType = Base.extend({

     /\*\*

      \* @pulic

      \* @returns {String}

      \*/

     foo: function(){ return "foo public"; },

     /\*\*

      \* @constructs SuperType

      \*/

     constructor: function () {}

   }),

   /\*\*

    \* @class

    \*/

   Constructor = SuperType.extend({

     /\*\*

      \* @pulic

      \* @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 (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 like class-based, but in fact this a syntactic sugar over existing prototypes. You can check  with typeof ConcreteClass and it will give you "function" because ConcreteClass is a canonical constructor. So we need no trick anymore for extending supertypes, no trick to refer 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. That is still in discussion for ES7 (https://esdiscuss.org/topic/es7-property-initializers). Besides this we can declare straight in the body of a class its static methods:

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

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

# How to: magic methods in JavaScript

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

"use strict";

var bar = {

 /\*\* @type {[Number]} \*/

 arr: [ 1, 2 ],

 /\*\*

  \* 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 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'" );

 }

});

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'

Object.defineProperty as well as the second parameter of Object.create specifies property configuration (whether it is enumerable, configurable, immutable, how it can be accessed or modified). So we can achieve a similar effect by configuring the property as read-only:

"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:

delete bar.length;

## Accessors in ES6 classes

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

"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:

"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 showing access control to known properties. But if I want a custom storage with variadic interface similar to localStorage. A storage with getItem method to retrieve stored values and setItem method to set them. Besides this must work the same when you directly access or set the a pseudo-property (val = storage.aKey and storage.aKey = "value"). This can be achieved by using ES6 Proxy:

"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 JavaScript core features for maximum effect.