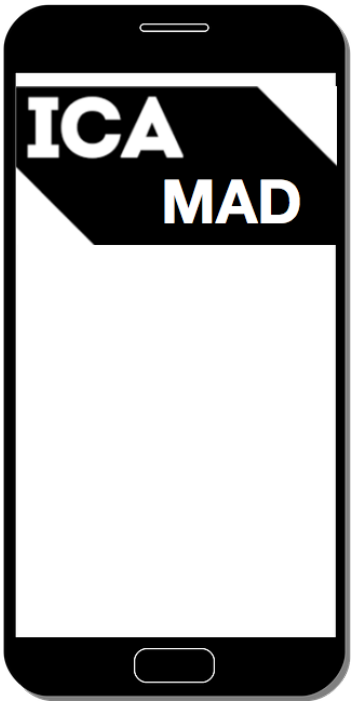

Develop a Hybrid App

Reader

Versie: 0.1 (concept)
Samenstelling: Bart van der Wal, september 2017

Inhoudsopgave

JavaScript The Good Parts – Douglas Crockford
- Hoofdstuk 1, 3 en stukje 4
- TypeScript DeepDive Basarat – TODO
- Building Apps with Ionic – Chris Griffith - TODO

Unearthing the Excellence in JavaScript

JavaScript: The Good Parts

O'REILLY® | YAHOO! PRESS

Douglas Crockford

JavaScript: The Good Parts

Douglas Crockford

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Table of Contents

CHAPTER 1
Good Parts

…setting the attractions of my
good parts aside I have no other charms.
—William Shakespeare, *The Merry Wives of Windsor*

When I was a young journeyman programmer, I would learn about every feature of the languages I was using, and I would attempt to use all of those features when I wrote. I suppose it was a way of showing off, and I suppose it worked because I was the guy you went to if you wanted to know how to use a particular feature.

Eventually I figured out that some of those features were more trouble than they were worth. Some of them were poorly specified, and so were more likely to cause portability problems. Some resulted in code that was difficult to read or modify. Some induced me to write in a manner that was too tricky and error-prone. And some of those features were design errors. Sometimes language designers make mistakes.

Most programming languages contain good parts and bad parts. I discovered that I could be a better programmer by using only the good parts and avoiding the bad parts. After all, how can you build something good out of bad parts?

It is rarely possible for standards committees to remove imperfections from a language because doing so would cause the breakage of all of the bad programs that depend on those bad parts. They are usually powerless to do anything except heap more features on top of the existing pile of imperfections. And the new features do not always interact harmoniously, thus producing more bad parts.

But *you* have the power to define your own subset. You can write better programs by relying exclusively on the good parts.

JavaScript is a language with more than its share of bad parts. It went from nonexistence to global adoption in an alarmingly short period of time. It never had an interval in the lab when it could be tried out and polished. It went straight into Netscape Navigator 2 just as it was, and it was very rough. When Java™ applets failed, JavaScript became the "Language of the Web" by default. JavaScript's popularity is almost completely independent of its qualities as a programming language.

Fortunately, JavaScript has some extraordinarily good parts. In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of JavaScript is so effectively hidden that for many years the prevailing opinion of JavaScript was that it was an unsightly, incompetent toy. My intention here is to expose the goodness in JavaScript, an outstanding, dynamic programming language. JavaScript is a block of marble, and I chip away the features that are not beautiful until the language's true nature reveals itself. I believe that the elegant subset I carved out is vastly superior to the language as a whole, being more reliable, readable, and maintainable.

This book will not attempt to fully describe the language. Instead, it will focus on the good parts with occasional warnings to avoid the bad. The subset that will be described here can be used to construct reliable, readable programs small and large. By focusing on just the good parts, we can reduce learning time, increase robustness, and save some trees.

Perhaps the greatest benefit of studying the good parts is that you can avoid the need to unlearn the bad parts. Unlearning bad patterns is very difficult. It is a painful task that most of us face with extreme reluctance. Sometimes languages are subsetted to make them work better for students. But in this case, I am subsetting JavaScript to make it work better for professionals.

Why JavaScript?

JavaScript is an important language because it is the language of the web browser. Its association with the browser makes it one of the most popular programming languages in the world. At the same time, it is one of the most despised programming languages in the world. The API of the browser, the Document Object Model (DOM) is quite awful, and JavaScript is unfairly blamed. The DOM would be painful to work with in any language. The DOM is poorly specified and inconsistently implemented. This book touches only very lightly on the DOM. I think writing a *Good Parts* book about the DOM would be extremely challenging.

JavaScript is most despised because it isn't SOME OTHER LANGUAGE. If you are good in SOME OTHER LANGUAGE and you have to program in an environment that only supports JavaScript, then you are forced to use JavaScript, and that is annoying. Most people in that situation don't even bother to learn JavaScript first, and then they are surprised when JavaScript turns out to have significant differences from the SOME OTHER LANGUAGE they would rather be using, and that those differences matter.

The amazing thing about JavaScript is that it is possible to get work done with it without knowing much about the language, or even knowing much about programming. It is a language with enormous expressive power. It is even better when you know what you're doing. Programming is difficult business. It should never be undertaken in ignorance.

Analyzing JavaScript

JavaScript is built on some very good ideas and a few very bad ones.

The very good ideas include functions, loose typing, dynamic objects, and an expressive object literal notation. The bad ideas include a programming model based on global variables.

JavaScript's functions are first class objects with (mostly) lexical scoping. JavaScript is the first lambda language to go mainstream. Deep down, JavaScript has more in common with Lisp and Scheme than with Java. It is Lisp in C's clothing. This makes JavaScript a remarkably powerful language.

The fashion in most programming languages today demands strong typing. The theory is that strong typing allows a compiler to detect a large class of errors at compile time. The sooner we can detect and repair errors, the less they cost us. JavaScript is a loosely typed language, so JavaScript compilers are unable to detect type errors. This can be alarming to people who are coming to JavaScript from strongly typed languages. But it turns out that strong typing does not eliminate the need for careful testing. And I have found in my work that the sorts of errors that strong type checking finds are not the errors I worry about. On the other hand, I find loose typing to be liberating. I don't need to form complex class hierarchies. And I never have to cast or wrestle with the type system to get the behavior that I want.

JavaScript has a very powerful object literal notation. Objects can be created simply by listing their components. This notation was the inspiration for JSON, the popular data interchange format. (There will be more about JSON in Appendix E.)

A controversial feature in JavaScript is prototypal inheritance. JavaScript has a class-free object system in which objects inherit properties directly from other objects. This is really powerful, but it is unfamiliar to classically trained programmers. If you attempt to apply classical design patterns directly to JavaScript, you will be frustrated. But if you learn to work with JavaScript's prototypal nature, your efforts will be rewarded.

JavaScript is much maligned for its choice of key ideas. For the most part, though, those choices were good, if unusual. But there was one choice that was particularly bad: JavaScript depends on global variables for linkage. All of the top-level variables of all compilation units are tossed together in a common namespace called *the global object*. This is a bad thing because global variables are evil, and in JavaScript they are fundamental. Fortunately, as we will see, JavaScript also gives us the tools to mitigate this problem.

In a few cases, we can't ignore the bad parts. There are some unavoidable awful parts, which will be called out as they occur. They will also be summarized in Appendix A. But we will succeed in avoiding most of the bad parts in this book, summarizing much of what was left out in Appendix B. If you want to learn more about the bad parts and how to use them badly, consult any other JavaScript book.

The standard that defines JavaScript (aka JScript) is the third edition of *The ECMAScript Programming Language*, which is available from *http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf*. The language described in this book is a proper subset of ECMAScript. This book does not describe the whole language because it leaves out the bad parts. The treatment here is not exhaustive. It avoids the edge cases. You should, too. There is danger and misery at the edges.

Appendix C describes a programming tool called JSLint, a JavaScript parser that can analyze a JavaScript program and report on the bad parts that it contains. JSLint provides a degree of rigor that is generally lacking in JavaScript development. It can give you confidence that your programs contain only the good parts.

JavaScript is a language of many contrasts. It contains many errors and sharp edges, so you might wonder, "Why should I use JavaScript?" There are two answers. The first is that you don't have a choice. The Web has become an important platform for application development, and JavaScript is the only language that is found in all browsers. It is unfortunate that Java failed in that environment; if it hadn't, there could be a choice for people desiring a strongly typed classical language. But Java did fail and JavaScript is flourishing, so there is evidence that JavaScript did something right.

The other answer is that, despite its deficiencies, *JavaScript is really good*. It is lightweight and expressive. And once you get the hang of it, functional programming is a lot of fun.

But in order to use the language well, you must be well informed about its limitations. I will pound on those with some brutality. Don't let that discourage you. The good parts are good enough to compensate for the bad parts.

A Simple Testing Ground

If you have a web browser and any text editor, you have everything you need to run JavaScript programs. First, make an HTML file with a name like *program.html*:

```
<html><body><pre><script src="program.js">
</script></pre></body></html>
```

Then, make a file in the same directory with a name like *program.js*:

```
document.writeln('Hello, world!');
```

Next, open your HTML file in your browser to see the result. Throughout the book, a `method` method is used to define new methods. This is its definition:

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

It will be explained in Chapter 4.

CHAPTER 3

Objects

Upon a homely object Love can wink.
—William Shakespeare, *The Two Gentlemen of Verona*

The simple types of JavaScript are numbers, strings, booleans (`true` and `false`), `null`, and `undefined`. All other values are *objects*. Numbers, strings, and booleans are object-like in that they have methods, but they are immutable. Objects in JavaScript are mutable keyed collections. In JavaScript, arrays are objects, functions are objects, regular expressions are objects, and, of course, objects are objects.

An object is a container of properties, where a property has a name and a value. A property name can be any string, including the empty string. A property value can be any JavaScript value except for `undefined`.

Objects in JavaScript are class-free. There is no constraint on the names of new properties or on the values of properties. Objects are useful for collecting and organizing data. Objects can contain other objects, so they can easily represent tree or graph structures.

JavaScript includes a prototype linkage feature that allows one object to inherit the properties of another. When used well, this can reduce object initialization time and memory consumption.

Object Literals

Object literals provide a very convenient notation for creating new object values. An object literal is a pair of curly braces surrounding zero or more name/value pairs. An object literal can appear anywhere an expression can appear:

```
var empty_object = {};

var stooge = {
    "first-name": "Jerome",
    "last-name": "Howard"
};
```

A property's name can be any string, including the empty string. The quotes around a property's name in an object literal are optional if the name would be a legal JavaScript name and not a reserved word. So quotes are required around `"first-name"`, but are optional around `first_name`. Commas are used to separate the pairs.

A property's value can be obtained from any expression, including another object literal. Objects can nest:

```
var flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

Retrieval

Values can be retrieved from an object by wrapping a string expression in a `[ ]` suffix. If the string expression is a constant, and if it is a legal JavaScript name and not a reserved word, then the `.` notation can be used instead. The `.` notation is preferred because it is more compact and it reads better:

```
stooge["first-name"]     // "Joe"
flight.departure.IATA    // "SYD"
```

The `undefined` value is produced if an attempt is made to retrieve a nonexistent member:

```
stooge["middle-name"]    // undefined
flight.status            // undefined
stooge["FIRST-NAME"]     // undefined
```

The `||` operator can be used to fill in default values:

```
var middle = stooge["middle-name"] || "(none)";
var status = flight.status || "unknown";
```

Attempting to retrieve values from `undefined` will throw a `TypeError` exception. This can be guarded against with the `&&` operator:

```
flight.equipment                              // undefined
flight.equipment.model                        // throw "TypeError"
flight.equipment && flight.equipment.model    // undefined
```

Update

A value in an object can be updated by assignment. If the property name already exists in the object, the property value is replaced:

```
stooge['first-name'] = 'Jerome';
```

If the object does not already have that property name, the object is augmented:

```
stooge['middle-name'] = 'Lester';
stooge.nickname = 'Curly';
flight.equipment = {
    model: 'Boeing 777'
};
flight.status = 'overdue';
```

Reference

Objects are passed around by reference. They are never copied:

```
var x = stooge;
x.nickname = 'Curly';
var nick = stooge.nickname;
    // nick is 'Curly' because x and stooge
    // are references to the same object

var a = {}, b = {}, c = {};
    // a, b, and c each refer to a
    // different empty object
a = b = c = {};
    // a, b, and c all refer to
    // the same empty object
```

Prototype

Every object is linked to a prototype object from which it can inherit properties. All objects created from object literals are linked to `Object.prototype`, an object that comes standard with JavaScript.

When you make a new object, you can select the object that should be its prototype. The mechanism that JavaScript provides to do this is messy and complex, but it can be significantly simplified. We will add a `create` method to the `Object` function. The `create` method creates a new object that uses an old object as its prototype. There will be much more about functions in the next chapter.

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

```
}
var another_stooge = Object.create(stooge);
```

The prototype link has no effect on updating. When we make changes to an object, the object's prototype is not touched:

```
another_stooge['first-name'] = 'Harry';
another_stooge['middle-name'] = 'Moses';
another_stooge.nickname = 'Moe';
```

The prototype link is used only in retrieval. If we try to retrieve a property value from an object, and if the object lacks the property name, then JavaScript attempts to retrieve the property value from the prototype object. And if that object is lacking the property, then it goes to *its* prototype, and so on until the process finally bottoms out with `Object.prototype`. If the desired property exists nowhere in the prototype chain, then the result is the `undefined` value. This is called *delegation*.

The prototype relationship is a dynamic relationship. If we add a new property to a prototype, that property will immediately be visible in all of the objects that are based on that prototype:

```
stooge.profession = 'actor';
another_stooge.profession    // 'actor'
```

We will see more about the prototype chain in Chapter 6.

Reflection

It is easy to inspect an object to determine what properties it has by attempting to retrieve the properties and examining the values obtained. The `typeof` operator can be very helpful in determining the type of a property:

```
typeof flight.number      // 'number'
typeof flight.status      // 'string'
typeof flight.arrival     // 'object'
typeof flight.manifest    // 'undefined'
```

Some care must be taken because any property on the prototype chain can produce a value:

```
typeof flight.toString    // 'function'
typeof flight.constructor // 'function'
```

There are two approaches to dealing with these undesired properties. The first is to have your program look for and reject function values. Generally, when you are reflecting, you are interested in data, and so you should be aware that some values could be functions.

The other approach is to use the `hasOwnProperty` method, which returns `true` if the object has a particular property. The `hasOwnProperty` method does not look at the prototype chain:

```
flight.hasOwnProperty('number')         // true
flight.hasOwnProperty('constructor')    // false
```

Enumeration

The `for in` statement can loop over all of the property names in an object. The enumeration will include all of the properties—including functions and prototype properties that you might not be interested in—so it is necessary to filter out the values you don't want. The most common filters are the `hasOwnProperty` method and using `typeof` to exclude functions:

```
var name;
for (name in another_stooge) {
    if (typeof another_stooge[name] !== 'function') {
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

There is no guarantee on the order of the names, so be prepared for the names to appear in any order. If you want to assure that the properties appear in a particular order, it is best to avoid the `for in` statement entirely and instead make an array containing the names of the properties in the correct order:

```
var i;
var properties = [
    'first-name',
    'middle-name',
    'last-name',
    'profession'
];
for (i = 0; i < properties.length; i += 1) {
    document.writeln(properties[i] + ': ' +
            another_stooge[properties[i]]);
}
```

By using `for` instead of `for in`, we were able to get the properties we wanted without worrying about what might be dredged up from the prototype chain, and we got them in the correct order.

Delete

The `delete` operator can be used to remove a property from an object. It will remove a property from the object if it has one. It will not touch any of the objects in the prototype linkage.

Removing a property from an object may allow a property from the prototype linkage to shine through:

```
another_stooge.nickname    // 'Moe'

// Remove nickname from another_stooge, revealing
// the nickname of the prototype.
```

```
delete another_stooge.nickname;

another_stooge.nickname    // 'Curly'
```

Global Abatement

JavaScript makes it easy to define global variables that can hold all of the assets of your application. Unfortunately, global variables weaken the resiliency of programs and should be avoided.

One way to minimize the use of global variables is to create a single global variable for your application:

```
var MYAPP = {};
```

That variable then becomes the container for your application:

```
MYAPP.stooge = {
    "first-name": "Joe",
    "last-name": "Howard"
};

MYAPP.flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

By reducing your global footprint to a single name, you significantly reduce the chance of bad interactions with other applications, widgets, or libraries. Your program also becomes easier to read because it is obvious that `MYAPP.stooge` refers to a top-level structure. In the next chapter, we will see ways to use closure for information hiding, which is another effective global abatement technique.

CHAPTER 4

Functions

Why, every fault's condemn'd ere it be done:
Mine were the very cipher of a function…
—William Shakespeare, *Measure for Measure*

The best thing about JavaScript is its implementation of functions. It got almost everything right. But, as you should expect with JavaScript, it didn't get everything right.

A function encloses a set of statements. Functions are the fundamental modular unit of JavaScript. They are used for code reuse, information hiding, and composition. Functions are used to specify the behavior of objects. Generally, the craft of programming is the factoring of a set of requirements into a set of functions and data structures.

Function Objects

Functions in JavaScript are objects. Objects are collections of name/value pairs having a hidden link to a prototype object. Objects produced from object literals are linked to `Object.prototype`. Function objects are linked to `Function.prototype` (which is itself linked to `Object.prototype`). Every function is also created with two additional hidden properties: the function's context and the code that implements the function's behavior.

Every function object is also created with a `prototype` property. Its value is an object with a `constructor` property whose value is the function. This is distinct from the hidden link to `Function.prototype`. The meaning of this convoluted construction will be revealed in the next chapter.

Since functions are objects, they can be used like any other value. Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to functions, and functions can be returned from functions. Also, since functions are objects, functions can have methods.

The thing that is special about functions is that they can be invoked.

Function Literal

Function objects are created with function literals:

```
// Create a variable called add and store a function
// in it that adds two numbers.

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

A function literal has four parts. The first part is the reserved word `function`.

The optional second part is the function's name. The function can use its name to call itself recursively. The name can also be used by debuggers and development tools to identify the function. If a function is not given a name, as shown in the previous example, it is said to be *anonymous*.

The third part is the set of parameters of the function, wrapped in parentheses. Within the parentheses is a set of zero or more parameter names, separated by commas. These names will be defined as variables in the function. Unlike ordinary variables, instead of being initialized to `undefined`, they will be initialized to the arguments supplied when the function is invoked.

The fourth part is a set of statements wrapped in curly braces. These statements are the body of the function. They are executed when the function is invoked.

A function literal can appear anywhere that an expression can appear. Functions can be defined inside of other functions. An inner function of course has access to its parameters and variables. An inner function also enjoys access to the parameters and variables of the functions it is nested within. The function object created by a function literal contains a link to that outer context. This is called *closure*. This is the source of enormous expressive power.

Invocation

Invoking a function suspends the execution of the current function, passing control and parameters to the new function. In addition to the declared parameters, every function receives two additional parameters: `this` and `arguments`. The `this` parameter is very important in object oriented programming, and its value is determined by the *invocation pattern*. There are four patterns of invocation in JavaScript: the method invocation pattern, the function invocation pattern, the constructor invocation pattern, and the apply invocation pattern. The patterns differ in how the bonus parameter `this` is initialized.

CHAPTER 10
Beautiful Features

Thus, expecting thy reply, I profane my lips on thy
foot, my eyes on thy picture, and my heart on thy
every part. Thine, in the dearest design of industry…
—William Shakespeare, *Love's Labor's Lost*

I was invited last year to contribute a chapter to Andy Oram's and Greg Wilson's *Beautiful Code* (O'Reilly), an anthology on the theme of beauty as expressed in computer programs. I wanted to write my chapter in JavaScript. I wanted to use it to present something abstract, powerful, and useful to show that the language was up to it. And I wanted to avoid the browser and other venues in which JavaScript is typecast. I wanted to show something respectable with some heft to it.

I immediately thought of Vaughn Pratt's Top Down Operator Precedence parser, which I use in JSLint (see Appendix C). Parsing is an important topic in computing. The ability to write a compiler for a language in itself is still a test for the completeness of a language.

I wanted to include all of the code for a parser in JavaScript that parses JavaScript. But my chapter was just one of 30 or 40, so I felt constrained in the number of pages I could consume. A further complication was that most of my readers would have no experience with JavaScript, so I also would have to introduce the language and its peculiarities.

So, I decided to subset the language. That way, I wouldn't have to parse the whole language, and I wouldn't have to describe the whole language. I called the subset Simplified JavaScript. Selecting the subset was easy: it included just the features that I needed to write a parser. This is how I described it in *Beautiful Code*:

> Simplified JavaScript is just the good stuff, including:
>
> *Functions as first class objects*
> Functions in Simplified JavaScript are lambdas with lexical scoping.

Dynamic objects with prototypal inheritance
: Objects are class-free. We can add a new member to any object by ordinary assignment. An object can inherit members from another object.

Object literals and array literals
: This is a very convenient notation for creating new objects and arrays. JavaScript literals were the inspiration for the JSON data interchange format.

The subset contained the best of the Good Parts. Even though it was a small language, it was very expressive and powerful. JavaScript has lots of additional features that really don't add very much, and as you'll find in the appendixes that follow, it has a lot of features with negative value. There was nothing ugly or bad in the subset. All of that fell away.

Simplified JavaScript isn't strictly a subset. I added a few new features. The simplest was adding `pi` as a simple constant. I did that to demonstrate a feature of the parser. I also demonstrated a better reserved word policy and showed that reserved words are unnecessary. In a function, a word cannot be used as both a variable or parameter name and a language feature. You can use a word for one or the other, and the programmer gets to choose. That makes a language easier to learn because you don't need to be aware of features you don't use. And it makes the language easier to extend because it isn't necessary to reserve more words to add new features.

I also added block scope. Block scope is not a necessary feature, but not having it confuses experienced programmers. I included block scope because I anticipated that my parser would be used to parse languages that are not JavaScript, and those languages would do scoping correctly. The code I wrote for the parser is written in a style that doesn't care if block scope is available or not. I recommend that you write that way, too.

When I started thinking about this book, I wanted to take the subset idea further, to show how to take an existing programming language and make significant improvements to it by making no changes except to exclude the low-value features.

We see a lot of feature-driven product design in which the cost of features is not properly accounted. Features can have a negative value to consumers because they make the products more difficult to understand and use. We are finding that people like products that just work. It turns out that designs that just work are much harder to produce than designs that assemble long lists of features.

Features have a specification cost, a design cost, and a development cost. There is a testing cost and a reliability cost. The more features there are, the more likely one will develop problems or will interact badly with another. In software systems, there is a storage cost, which was becoming negligible, but in mobile applications is becoming significant again. There are ascending performance costs because Moore's Law doesn't apply to batteries.

Typed Wrappers

JavaScript has a set of typed wrappers. For example:

```
new Boolean(false)
```

produces an object that has a `valueOf` method that returns the wrapped value. This turns out to be completely unnecessary and occasionally confusing. Don't use `new Boolean` or `new Number` or `new String`.

Also avoid `new Object` and `new Array`. Use {} and [] instead.

new

JavaScript's `new` operator creates a new object that inherits from the operand's prototype member, and then calls the operand, binding the new object to `this`. This gives the operand (which had better be a constructor function) a chance to customize the new object before it is returned to the requestor.

If you forget to use the `new` operator, you instead get an ordinary function call, and `this` is bound to the global object instead of to a new object. That means that your function will be clobbering global variables when it attempts to initialize the new members. That is a very bad thing. There is no compile-time warning. There is no runtime warning.

By convention, functions that are intended to be used with `new` should be given names with initial capital letters, and names with initial capital letters should be used only with constructor functions that take the `new` prefix. This convention gives us a visual cue that can help spot expensive mistakes that the language itself is keen to overlook.

An even better coping strategy is to not use `new` at all.

void

In many languages, `void` is a type that has no values. In JavaScript, `void` is an operator that takes an operand and returns `undefined`. This is not useful, and it is very confusing. Avoid `void`.