



# JavaScript Summary

## What you need to know about JavaScript

Publication date: 8 November 2015

*A short summary of JavaScript's main features, including a discussion of the different kinds of JS objects, basic data structures, functions as first class citizens, and code patterns for implementing classes.*

By Gerd Wagner

[web-engineering.info](http://web-engineering.info)



Copyright 2015, Gerd Wagner. This article is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/). Available online at <http://web-engineering.info/>

[Summaries](#)[Cheatsheets](#)[Posters](#).

## Contents

- [Introduction](#)..... 1
- [Types and Data Literals](#) ..... 2
- [Variable Scope](#) ..... 4
- [Strict Mode](#)..... 4
- [Different Kinds of Objects](#)..... 5
- [Array Lists](#) ..... 6
- [Maps](#)..... 6
- [Four Types of Basic Data Structures](#) ..... 7
- [Methods and Functions](#) ..... 8
- [Defining and Using Classes](#) ..... 9
  - [Constructor-based classes](#) ..... 11
  - [Factory-based classes](#)..... 13
- [JavaScript as an Object-Oriented Language](#) ..... 14
- [The localStorage API](#)..... 14
- [Further Reading about JavaScript](#)..... 15

### 1 Introduction

JavaScript was developed in 10 days in May 1995 by [Brendan Eich](#), then working at [Netscape](#), as the HTML scripting language for their browser *Navigator 2* ([more about](#)

[history](#)). Brendan Eich said (at the O'Reilly Fluent conference in San Francisco in April 2015): "I did JavaScript in such a hurry, I never dreamed it would become the assembly language for the Web".

JavaScript is a dynamic functional object-oriented programming language that can be used for

1. Enriching a web page by
  - generating browser-specific HTML content or CSS styling,
  - inserting dynamic HTML content,
  - producing special audio-visual effects (animations).
2. Enriching a web user interface by
  - implementing advanced user interface components,
  - validating user input on the client side,
  - automatically pre-filling certain form fields.
3. Implementing a front-end web application with local or remote data storage, as described in the book [Building Front-End Web Apps with Plain JavaScript](#).
4. Implementing a front-end component for a distributed web application with remote data storage managed by a back-end component, which is a server-side program that is traditionally written in a server-side language such as PHP, Java or C#, but can nowadays also be written in JavaScript with NodeJS.
5. Implementing a complete distributed web application where both the front-end and the back-end components are JavaScript programs.

The version of JavaScript that is currently supported by web browsers is called "ECMAScript 5.1", or simply "ES5", but the next two versions, called "ES6" and "ES7" (or "ES 2015" and "ES 2016", as new versions are planned on a yearly basis), with lots of added functionality and improved syntaxes, are around the corner (and already partially supported by current browsers and back-end JS environments).

This article has been extracted from the book [Building Front-End Web Apps with Plain JavaScript](#), which is available as an open access online book. It tries to take all impor-

tant points of the [classical JavaScript summary](#) by [Douglas Crockford](#) into consideration.

## 2 Types and Data Literals

JavaScript has three primitive data types: `string`, `number` and `boolean`, and we can test if a variable `v` holds a value of such a type with the help of `typeof(v)` as, for instance, in `typeof(v) === "number"`.

There are five basic reference types: `Object`, `Array`, `Function`, `Date` and `RegExp`. Arrays and functions are just special kinds of objects, but, conceptually, dates and regular expressions are primitive data values, and happen to be implemented in the form of wrapper objects.

The types of variables, array elements, function parameters and return values are not declared and are normally not checked by JavaScript engines. Type conversion (casting) is performed automatically.

The value of a variable may be

1. a *data value*: either a string, a number, or a boolean;
2. an *object reference*: either referencing an ordinary object, or an array, function, date, or regular expression;
3. the special data value `null`, which is typically used as a default value for initializing an object variable;
4. the special data value `undefined`, which is the implicit initial value of all variables that have been declared but not initialized.

A *string* is a sequence of Unicode characters. String literals, like "Hello world!", 'A3F0', or the empty string "", are enclosed in single or double quotes. Two string expressions can be concatenated with the `+` operator, and checked for equality with the triple equality operator:

```
if (firstName + lastName === "James Bond") ...
```

The number of characters of a string can be obtained by applying the `length` attribute to a string:

```
console.log( "Hello world!".length); // 12
```

All *numeric* data values are represented in 64-bit floating point format with an optional exponent (like in the numeric data literal

3.1e10). There is no explicit type distinction between integers and floating point numbers. If a numeric expression cannot be evaluated to a number, its value is set to `NaN` ("not a number"), which can be tested with the built-in predicate `isNaN(expr)`.

Unfortunately, a built-in function, `Number.isInteger`, for testing if a number is an *integer* has only been introduced in ES6, so a [polyfill](#) is needed for using it in browsers that do not yet support it. For making sure that a numeric value is an integer, or that a string representing a number is converted to an integer, one has to apply the predefined function `parseInt`. Similarly, a string representing a decimal number can be converted to this number with `parseFloat`. For converting a number `n` to a string, the best method is using `String(n)`.

Like in Java, there are two pre-defined *Boolean* data literals, `true` and `false`, and the Boolean operator symbols are the exclamation mark `!` for NOT, the double ampersand `&&` for AND, and the double bar `||` for OR. When a non-Boolean value is used in a condition, or as an operand of a Boolean expression, it is implicitly converted into a Boolean value according to the following rules. The empty string, the (numerical) data literal `0`, as well as `undefined` and `null`, are mapped to *false*, and all other values are mapped to *true*. This conversion can be performed explicitly with the help of the double negation operation `!!`.

In addition to strings, numbers and Boolean values, also *calendar dates* and times are important types of primitive data values, although they are not implemented as primitive values, but in the form of wrapper objects instantiating `Date`. Notice that `Date` objects do, in fact, not really represent dates, but

rather date-time instants represented internally as the number of milliseconds since 1 January, 1970 UTC. For converting the internal value of a `Date` object to a human-readable string, we have several options. The two most important options are using either the standard format of ISO date/time strings of the form "2015-01-27", or the format of localized date/time strings like "27.1.2015" (for simplicity, we have omitted the time part of the date/time strings in these examples). When `x instanceof Date`, then `x.toISOString()` provides the ISO date/time string, and `x.toLocaleDateString()` provides the localized date/time string. Given any date string `ds`, ISO or localized, `new Date(ds)` creates a corresponding date object.

For *equality* and inequality testing, always use the triple equality symbols `===` and `!==` instead of the double equality symbols `==` and `!=`. Otherwise, for instance, the number `2` would be the same as the string `"2"`, since the condition `(2 == "2")` evaluates to *true* in JavaScript.

Assigning an *empty array literal*, as in `var a = []` is the same as, but more concise than and therefore preferred to, invoking the `Array()` constructor without arguments, as in `var a = new Array()`.

Assigning an *empty object literal*, as in `var o = {}` is the same as, but more concise than and therefore preferred to, invoking the `Object()` constructor without arguments, as in `var o = new Object()`. Notice, however, that an empty object literal `{}` is not really empty, as it contains property slots and method slots inherited from `Object.prototype`. So, a truly empty object (without any slots) has to be created with `null` as prototype, like in `var emptyObject = Object.create(null)`.

Table 1: Type testing

Type	Example values	Test if <code>x</code> of type
string	"Hello world!", 'A3F0'	<code>typeof(x) === "string"</code>
boolean	true, false	<code>typeof(x) === "boolean"</code>

Type	Example values	Test if x of type
(floating point) number	-2.75, 0, 1, 1.0, 3.1e10	<code>typeof(x) === "number"</code>
integer	-2, 0, 1, 250	<code>Number.isInteger(x) *</code>
Object	<code>{}</code> , <code>{num:3, denom:4}</code> , <code>{isbn:"006251587X," title:"Weaving the Web"}</code> , <code>{"one":1, "two":2, "three":3}</code>	excluding null: x <code>instanceof Object</code> including null: <code>typeof(x) === "object"</code>
Array	<code>[]</code> , <code>["one"]</code> , <code>[1,2,3]</code> , <code>[1,"one", {}]</code>	<code>Array.isArray(x)</code>
Function	<code>function () { return "one"+1;}</code>	<code>typeof(x) === "function"</code>
Date	<code>new Date("2015-01-27")</code>	<code>x instanceof Date</code>
RegExp	<code>/(\w+)\s(\w+)/</code>	<code>x instanceof RegExp</code>

Table 2: Type conversions

Type	Convert to string	Convert string to type
boolean	<code>String(x)</code>	<code>Boolean(y)</code>
(floating point) number	<code>String(x)</code>	<code>parseFloat(y)</code>
integer	<code>String(x)</code>	<code>parseInt(y)</code>
Object	<code>x.toString()</code> OR <code>JSON.stringify(x)</code>	<code>JSON.parse(y)</code>
Array	<code>x.toString()</code> OR <code>JSON.stringify(x)</code>	<code>y.split()</code> OR <code>JSON.parse(y)</code>
Function	<code>x.toString()</code>	<code>new Function(y)</code>
Date	<code>x.toISOString()</code>	<code>new Date(y)</code>
RegExp	<code>x.toString()</code>	<code>new RegExp(y)</code>

### 3 Variable Scope

In the current version of JavaScript, ES5, there are only two kinds of scope for variables: the global scope (with `window` as the context object) and function scope, but **no block scope**. Consequently, declaring a variable within a block is confusing and should be avoided. For instance, although this is a frequently used pattern, even by experienced JavaScript programmers, it is a pitfall to declare the counter variable of a `for` loop in the loop, as in

```
function foo() {
  for (var i=0; i < 10; i++) {
    ... // do something with i
  }
}
```

Instead, and this is exactly how JavaScript is interpreting this code (by means of "hoisting" variable declarations), we should write:

```
function foo() {
  var i=0;
  for (i=0; i < 10; i++) {
    ... // do something with i
  }
}
```

All variables should be declared at the beginning of a function. Only in the next version of JavaScript, ES6, block scope will be supported by means of a new form of variable declaration with the keyword `let`.

### 4 Strict Mode

Starting from ES5, we can use [strict mode](#) for getting more runtime error checking. For in-

stance, in strict mode, all variables must be declared. An assignment to an undeclared variable throws an exception.

We can turn strict mode on by typing the following statement as the first line in a JavaScript file or inside a `<script>` element:

```
'use strict';
```

It is generally recommended that you use strict mode, except your code depends on libraries that are incompatible with strict mode.

## 5 Different Kinds of Objects

JS objects are different from classical OO/UML objects. In particular, they *need not instantiate a class*. And they can have their own (instance-level) methods in the form of method slots, so they do not only have (ordinary) *property slots*, but also *method slots*. In addition they may also have *key-value slots*. So, they may have three different kinds of slots, while classical objects only have property slots.

A JS object is essentially a set of name-value-pairs, also called *slots*, where names can be *property names*, *function names* or *keys* of a map. Objects can be created in an ad-hoc manner, using JavaScript's object literal notation (JSON), without instantiating a class:

```
var person1 = { firstName:"Tom",
                lastName:"Smith"};
```

Whenever the name in a slot is an [admissible JavaScript identifier](#), the slot may be either a *property slot*, a *method slot* or a *key-value slot*. Otherwise, if the name is some other type of string (in particular when it contains any blank space), then the slot represents a *key-value slot*, which is a map element, as explained below.

The name in a *property slot* may denote either

1. a *data-valued property*, in which case the value is a *data value* or, more generally, a *data-valued expression*; or
2. an *object-valued property*, in which case the value is an *object reference* or, more generally, an *object expression*.

The name in a *method slot* denotes a *JS function* (better called *method*), and its value is a *JS function definition expression*.

Object properties can be accessed in two ways:

1. Using the dot notation (like in C++/Java):

```
person1.lastName = "Smith"
```

2. Using a map notation:

```
person1["lastName"] = "Smith"
```

JS objects can be used in many different ways for different purposes. Here are five different use cases for, or possible meanings of, JS objects:

1. A *record* is a set of property slots like, for instance,

```
var myRecord = { firstName:"Tom",
                lastName:"Smith", age:26}
```

2. A *map* (also called 'associative array', 'dictionary', 'hash map' or 'hash table' in other languages) supports look-ups of *values* based on *keys* like, for instance,

```
var numeral2number = {"one":"1",
                      "two":"2", "three":"3"}
```

which associates the value "1" with the key "one", "2" with "two", etc. A key need not be a valid JavaScript identifier, but can be any kind of string (e.g. it may contain blank spaces).

3. An *untyped object* does not instantiate a class. It may have property slots and method slots like, for instance,

```
var person1 = {
  lastName: "Smith",
  firstName: "Tom",
  getFullName: function () {
    return this.firstName + " " +
           this.lastName;
  }
};
```

Within the body of a method slot of an object, the special variable `this` refers to the object.

4. A *namespace* may be defined in the form of an untyped object referenced by a global object variable, the name of which represents a namespace prefix. For instance, the following object variable provides the main namespace of an application based on the *Model-View-Controller (MVC)* ar-

chitecture paradigm where we have three sub-namespaces corresponding to the three parts of an MVC application:

```
var myApp = { model:{},
              view:{}, ctrl:{} };
```

A more advanced namespace mechanism can be obtained by using an immediately invoked JS function expression, as explained below.

5. A **typed object** instantiates a class that is defined either by a JavaScript constructor function or by a factory object. See the section [Defining and using classes](#) below

## 6 Array Lists

A JavaScript array represents, in fact, the logical data structure of an *array list*, which is a list where each list item can be accessed via an index number (like the elements of an array). Using the term 'array' without saying 'JS array' creates a terminological ambiguity. But for simplicity, we will sometimes just say 'array' instead of 'JS array'.

A variable may be initialized with a JavaScript *array literal*:

```
var a = [1,2,3];
```

Because they are array lists, JS arrays can grow dynamically: it is possible to use indexes that are greater than the length of the array. For instance, after the array variable initialization above, the array held by the variable `a` has the length 3, but still we can assign a fifth array element like in

```
a[4] = 7;
```

The contents of an array `a` are processed with the help of a standard *for* loop with a counter variable counting from the first array index 0 to the last array index, which is `a.length-1`:

```
for (i=0; i < a.length; i++) { ... }
```

Since arrays are special types of objects, we sometimes need a method for finding out if a variable represents an array. We can test, if a variable `a` represents an array with `Array.isArray( a )`.

For **adding** a new element to an array, we append it to the array using the `push` operation as in:

```
a.push( newElement );
```

For **deleting** an element at position `i` from an array `a`, we use the pre-defined array method `splice` as in:

```
a.splice( i, 1 );
```

For **searching** a value `v` in an array `a`, we can use the pre-defined array method `indexOf`, which returns the position, if found, or -1, otherwise, as in:

```
if (a.indexOf(v) > -1) ...
```

For **looping** over an array `a`, we have two options: either use a *for* loop, or the array looping method `forEach`. In any case, we can use a *for* loop:

```
var i=0;
for (i=0; i < a.length; i++) {
  console.log( a[i] );
}
```

If performance doesn't matter, that is, if `a` is sufficiently small (say, it does not contain more than a few hundred elements), we can use the array looping method `forEach`, as in the following example, where the parameter `elem` iteratively assumes each element of the array `a` as its value:

```
a.forEach( function (elem) {
  console.log( elem );
})
```

For **cloning** an array `a`, we can use the array function `slice` in the following way:

```
var clone = a.slice(0);
```

## 7 Maps

A map (also called 'hash map' or 'associative array') provides a mapping from keys to their associated values. The keys of a JS map are string literals that may include blank spaces like in:

```
var myTranslation = {
  "my house": "mein Haus",
  "my boat": "mein Boot",
  "my horse": "mein Pferd"
}
```

A map is processed by looping over all keys of the map with the help of the pre-defined function `Object.keys( m )`, which returns an array of all keys of a map `m`. For instance,

```
var i=0, key="", keys=[];
keys = Object.keys( myTranslation);
for (i=0; i < keys.length; i++) {
  key = keys[i];
  alert('The translation of '+ key +' is '+
    myTranslation[key]);
}
```

For **adding** a new entry to a map, we simply associate the new value with its key as in:

```
myTranslation["my car"] = "mein Auto";
```

For **deleting** an entry from a map, we can use the pre-defined `delete` operator as in:

```
delete myTranslation["my boat"];
```

For **searching** in a map if it contains an entry for a certain key value, such as for testing if the translation map contains an entry for "my bike" we can check the following:

```
if ("my bike" in myTranslation) ...
```

For **looping** over a map `m`, we first convert it to an array of its keys with the help of the pre-defined `Object.keys` method, and then we can use either a `for` loop or the `forEach` method. The following example shows how to loop with `for`:

```
var i=0, key="", keys=[];
keys = Object.keys( m);
for (i=0; i < keys.length; i++) {
  key = keys[i];
  console.log( m[key]);
}
```

Again, if `m` is sufficiently small, we can use the `forEach` method, as in the following example:

```
Object.keys( m).forEach( function (key) {
  console.log( m[key]);
})
```

Notice that using the `forEach` method is more concise.

For **cloning** a map `m`, we can use the composition of `JSON.stringify` and `JSON.parse`.

We first serialize `m` to a string representation with `JSON.stringify`, and then de-serialize the string representation to a map object with `JSON.parse`:

```
var clone = JSON.parse( JSON.stringify( m))
```

Notice that this method works well if the map contains only simple data values or (possibly nested) arrays/maps containing simple data values. In other cases, e.g. if the map contains `Date` objects, we have to write our own clone method.

## 8 Four Types of Basic Data Structures

In summary, the four types of basic data structures supported are:

1. **array lists**, such as `["one", "two", "three"]`, which are special JS objects called 'arrays', but since they are dynamic, they are rather *array lists* as defined in the *Java* programming language.
2. **records**, which are special JS objects, such as `{firstName: "Tom", lastName: "Smith"}`, as discussed above,
3. **maps**, which are also special JS objects, such as `{"one":1, "two":2, "three":3}`, as discussed above,
4. **entity tables**, like for instance the Table 1 shown below, which are special maps where the values are entity records with a standard ID (or *primary key*) slot, such that the keys of the map are the standard IDs of these entity records.

Table 3: An entity table representing a collection of books

Key	Value
006251587X	{ isbn:"006251587X," title:"Weaving the Web", year:2000 }
0465026567	{ isbn:"0465026567," title:"Gödel, Escher, Bach", year:1999 }
0465030793	{ isbn:"0465030793," title:"I Am A Strange Loop", year:2008 }

Notice that our distinction between maps, records and entity tables is a purely conceptual distinction, and not a syntactical one. For a JavaScript engine, both `{firstName:"Tom",`

`lastName:"Smith"}` and `{"one":1, "two":2, "three":3}` are just objects. But conceptually, `{firstName:"Tom", lastName:"Smith"}` is a record because `firstName` and `lastName` are in-

tended to denote properties or fields, while `{"one":1, "two":2, "three":3}` is a map because "one" and "two" are not intended to denote properties/fields, but are just arbitrary string values used as keys for a map.

Making such conceptual distinctions helps to better understand the options offered by JavaScript.

## 9 Methods and Functions

In JavaScript, methods are called "functions", no matter if they return a value or not. As shown in Figure 1 below, JS functions are special JS objects, having an optional `name` property and a `length` property providing their number of parameters. If a variable `v` references a JS function can be tested with

```
if (typeof( v ) === "function") {...}
```

Being JS objects implies that JS functions can be stored in variables, passed as arguments to functions, returned by functions, have properties and can be changed dynamically. Therefore, functions are first-class citizens, and JavaScript can be viewed as a functional programming language,

The general form of a JS *function definition* is an assignment of a function expression to a variable:

```
var myF = function theNameOfMyF () {...}
```

where `theNameOfMyF` is optional. When it is omitted, the function is *anonymous*. In any case, functions are invoked via a variable that references the function. In the above case, this means that the function is invoked with `myF()`, and not with `theNameOfMyF()`.

Anonymous function expressions are called *lambda expressions* (or shorter *lambdas*) in other programming languages.

As an example of an anonymous function expression being passed as an argument in the invocation of another (higher-order) function, we can take a comparison function being passed to the pre-defined function `sort` for sorting the elements of an array list. Such a comparison function must return a negative number if its first argument is considered smaller than its second argument, it must return 0 if both arguments are of the same rank,

and it must return a positive number if the second argument is considered smaller than the first one. In the following example, we sort a list of lists of 2 numbers in lexicographic order:

```
var list = [[1,2],[1,3],[1,1],[2,1]];
list.sort( function (x,y) {
    return ((x[0] === y[0]) ?
        x[1]-y[1] : x[0]-y[0]);
});
```

A JS *function declaration* has the following form:

```
function theNameOfMyF () {...}
```

It is equivalent to the following named function definition:

```
var theNameOfMyF = function theNameOfMyF () {
    ...
}
```

that is, it creates both a function with name `theNameOfMyF` and a variable `theNameOfMyF` referencing this function.

JS functions can have *inner functions*. The *closure* mechanism allows a JS function using variables (except `this`) from its outer scope, and a function created in a closure remembers the environment in which it was created. In the following example, there is no need to pass the outer scope variable `result` to the inner function via a parameter, as it is readily available:

```
var sum = function (numbers) {
    var result = 0;
    numbers.forEach( function (n) {
        result += n;
    });
    return result;
};
console.log( sum([1,2,3,4]));
```

When a method/function is executed, we can access its arguments within its body by using the built-in `arguments` object, which is "array-like" in the sense that it has indexed elements and a `length` property, and we can iterate over it with a normal `for` loop, but since it's not an instance of `Array`, the JS array methods (such as the `forEach` looping method) cannot be applied to it. The `arguments` object contains an element for each argument passed to the method. This allows defining a method without parameters and invoking it with *any number of arguments*, like so:



```
var sum = function () {
  var result = 0, i=0;
  for (i=0; i < arguments.length; i++) {
    result = result + arguments[i];
  }
  return result;
};
console.log( sum(0,1,1,2,3,5,8)); // 20
```

A method defined on the prototype of a constructor function, which can be invoked on all objects created with that constructor, such as `Array.prototype.forEach`, where `Array` represents the constructor, has to be invoked with an instance of the class as **context object** referenced by the `this` variable (see also the next section on classes). In the following example, the array `numbers` is the context object in the invocation of `forEach`:

```
var numbers = [1,2,3]; // create an instance of Array
numbers.forEach( function (n) {
  console.log( n);
});
```

Whenever such a prototype method is to be invoked not with a context object, but with an object as an ordinary argument, we can do this with the help of **the JS function call method** that takes an object, on which the method is invoked, as its first parameter, followed by the parameters of the method to be invoked. For instance, we can apply the `forEach` looping method to the array-like object `arguments` in the following way:

```
var sum = function () {
  var result = 0;
  Array.prototype.forEach.call( arguments, function(n) {
    result = result + n;
  });
  return result;
};
```

A variant of the `Function.prototype.call` method, taking all arguments of the method to be invoked as a single array argument, is `Function.prototype.apply`.

Whenever a method defined for a prototype is to be invoked without a context object, or when a method defined in the context of an object is to be invoked without its context object, we can bind its `this` variable to a given object with the help of **the JS function bind method** (`Function.prototype.bind`). This allows creating a shortcut for invoking a method, as in `var querySel = document.querySelector.bind( document),`

which allows to use `querySel` instead of `document.querySelector`.

The option of immediately invoked JS function expressions can be used for obtaining a namespace mechanism that is superior to using a plain namespace object, since it can be controlled which variables and methods are globally exposed and which are not. This mechanism is also the basis for JS module concepts. In the following example, we define a namespace for the model code part of an app, which exposes some variables and the model classes in the form of constructor functions:

```
myApp.model = function () {
  var appName = "My app's name";
  var someNonExposedVariable = ...;
  function ModelClass1 () {...}
  function ModelClass2 () {...}
  function someNonExposedMethod (...) {...}
  return {
    appName: appName,
    ModelClass1: ModelClass1,
    ModelClass2: ModelClass2
  }
}(); // immediately invoked
```

This pattern has been proposed in the WebPlatform.org article [JavaScript best practices](#).

## 10 Defining and Using Classes

The concept of a **class** is fundamental in *object-oriented* programming. Objects *instantiate* (or *are classified by*) a class. A class defines the properties and methods (as a blueprint) for the objects created with it. Having a class concept is essential for being able to implement a *data model* in the form of **model classes** within a *Model-View-Controller (MVC)* architecture. However, classes and their inheritance/extension mechanism are over-used in classical OO languages, such as in Java, where all variables and procedures have to be defined in the context of a class and, consequently, classes are not only used for implementing object types (or model classes), but also as containers for many other purposes in these languages. This is not the case in JavaScript where we have the freedom to use classes for implementing object types only, while keeping method libraries in namespace objects.

Any code pattern for defining classes in JavaScript should satisfy five requirements. First of all, (1) it should allow to define a *class name*, a set of (instance-level) *properties*, preferably with the option to keep them 'private', a set of (instance-level) *methods*, and a set of *class-level properties and methods*. It's desirable that properties can be declared with a range/type, and with other meta-data, such as constraints. There should also be two introspection features: (2) an *is-instance-of predicate* that can be used for checking if an object is a direct or non-direct instance of a class, and (3) an instance-level property for retrieving the *direct type* of an object. In addition, it is desirable to have a third introspection feature for retrieving the *direct supertype* of a class. And finally, there should be two inheritance mechanisms: (4) *property inheritance* and (5) *method inheritance*. In addition, it is desirable to have support for *multiple inheritance* and *multiple classifications*, for allowing objects to play several roles at the same time by instantiating several role classes.

There is no explicit class concept in JavaScript. Different code patterns for defining classes in JavaScript have been proposed and are being used in different frameworks. But they do often not satisfy the five requirements listed above. The two most important approaches for defining classes are:

1. In the form of a *constructor* function that achieves method inheritance via the *prototype chain* and allows to create new instances of a class with the help of the `new` operator. This is the classical approach recommended by Mozilla in their [JavaScript Guide](#).
2. In the form of a *factory* object that uses the predefined `Object.create` method for creating new instances of a class. In this approach, the constructor-based inheritance mechanism has to be replaced by another mechanism. [Eric Elliott](#) has argued that factory-based classes are a viable alternative to constructor-based classes in JavaScript (in fact, he even condemns the use of classical inheritance with constructor-based classes, throwing out the baby with the bath water).

When building an app, we can use both types of classes, depending on the requirements of the app. Since we often need to define class hierarchies, and not just single classes, we have to make sure, however, that we don't mix these two alternative approaches within the same class hierarchy. While the factory-based approach, as exemplified by [mODELcLASSjs](#), has many advantages, which are summarized in Table 2, the constructor-based approach enjoys the advantage of higher performance object creation.

Table 4: Required and desirable features of JS code patterns for classes

Class feature	Constructor-based approach	Factory-based approach	<a href="#">mODELcLASSjs</a>
<b>Define properties and methods</b>	yes	yes	yes
Declare properties with a range (and other meta-data)	no	possibly	yes
<b>Built-in <i>is-instance-of</i> predicate</b>	yes	yes	yes
<b>Built-in <i>direct type</i> property</b>	yes	yes	yes
Built-in <i>direct supertype</i> property of classes	no	possibly	yes
<b>Property inheritance</b>	yes	yes	yes
<b>Method inheritance</b>	yes	yes	yes

Class feature	Constructor-based approach	Factory-based approach	mODELcLASSjs
Multiple inheritance	no	possibly	yes
Multiple classifications	no	possibly	yes
Allow <a href="#">object pools</a>	no	yes	yes

### 10.1 Constructor-based classes

Only in ES6, a user-friendly syntax for constructor-based classes has been introduced. In **Step 1.a)**, a base class `Person` is defined with two properties, `firstName` and `lastName`, as well as with an (instance-level) method `toString` and a static (class-level) method `checkLastName`:

```
class Person {
  constructor( first, last) {
    this.firstName = first;
    this.lastName = last;
  }
  toString() {
    return this.firstName + " " +
      this.lastName;
  }
  static checkLastName( ln) {
    if (typeof(ln)!="string" ||
        ln.trim()=== "") {
      console.log("Error: " +
        "invalid last name!");
    }
  }
}
```

In **Step 1.b)**, class-level ("static") properties are defined:

```
Person.instances = {};
```

Finally, in **Step 2)**, a subclass is defined with additional properties and methods that possibly override the corresponding superclass methods:

```
class Student extends Person {
  constructor( first, last, studNo) {
    super.constructor( first, last);
    this.studNo = studNo;
  }
  // method overrides superclass method
  toString() {
    return super.toString() + "(" +
      this.studNo + ")";
  }
}
```

In ES5, we can define a constructor-based class with a subclass in the form of constructor functions, following a code pattern recommended by Mozilla in their [JavaScript Guide](#), as shown in the following steps.

**Step 1.a)** First define the constructor function that implicitly defines the properties of the class by assigning them the values of the constructor parameters when a new object is created:

```
function Person( first, last) {
  this.firstName = first;
  this.lastName = last;
}
```

Notice that within a constructor, the special variable `this` refers to the new object that is created when the constructor is invoked.

**Step 1.b)** Next, define the *instance-level methods* of the class as method slots of the object referenced by the constructor's `prototype` property:

```
Person.prototype.toString = function () {
  return this.firstName + " " + this.lastName;
}
```

**Step 1.c)** *Class-level ("static") methods* can be defined as method slots of the constructor function itself (recall that, since JS functions are objects, they can have slots), as in

```
Person.checkLastName = function (ln) {
  if (typeof(ln)!="string" || ln.trim()=== "") {
    console.log("Error: invalid last name!");
  }
}
```

**Step 1.d)** Finally, define class-level ("static") properties as property slots of the constructor function:

```
Person.instances = {};
```

**Step 2.a):** Define a subclass with additional properties:

```
function Student( first, last, studNo) {
  // invoke superclass constructor
  Person.call( this, first, last);
  // define and assign additional properties
  this.studNo = studNo;
}
```

By invoking the supertype constructor with `Person.call( this, ...)` for any new object created, and referenced by `this`, as an instance of the subtype `Student`, we achieve that the property slots created in the supertype con-

structor (`firstName` and `lastName`) are also created for the subtype instance, along the entire chain of supertypes within a given class hierarchy. In this way we set up a **property inheritance** mechanism that makes sure that the own properties defined for an object on creation include the own properties defined by the supertype constructors.

In **Step 2b)**, we set up a mechanism for **method inheritance** via the constructor's `prototype` property. We assign a new object created from the supertype's `prototype` object to the `prototype` property of the subtype constructor and adjust the prototype's constructor property:

```
// Student inherits from Person
Student.prototype = Object.create(
    Person.prototype);
// adjust the subtype's constructor property
Student.prototype.constructor = Student;
```

With `Object.create( Person.prototype)` we create a new object with `Person.prototype` as its prototype and without any own property slots. By assigning this object to the `prototype` property of the subclass constructor, we achieve that the methods defined in, and inherited from, the superclass are also available for objects instantiating the subclass. This mechanism of chaining the prototypes takes care of method inheritance. Notice that setting `Student.prototype` to `Object.create( Person.prototype)` is preferable over setting it to `new Person()`, which was the way to achieve the same in the time before ES5.

**Step 2c):** Define a subclass method that overrides a superclass method:

```
Student.prototype.toString = function () {
    return Person.prototype.toString.call( this) +
        (" + this.studNo + ");
};
```

An instance of a constructor-based class is created by applying the `new` operator to the

constructor function and providing suitable arguments for the constructor parameters:

```
var pers1 = new Person("Tom", "Smith");
```

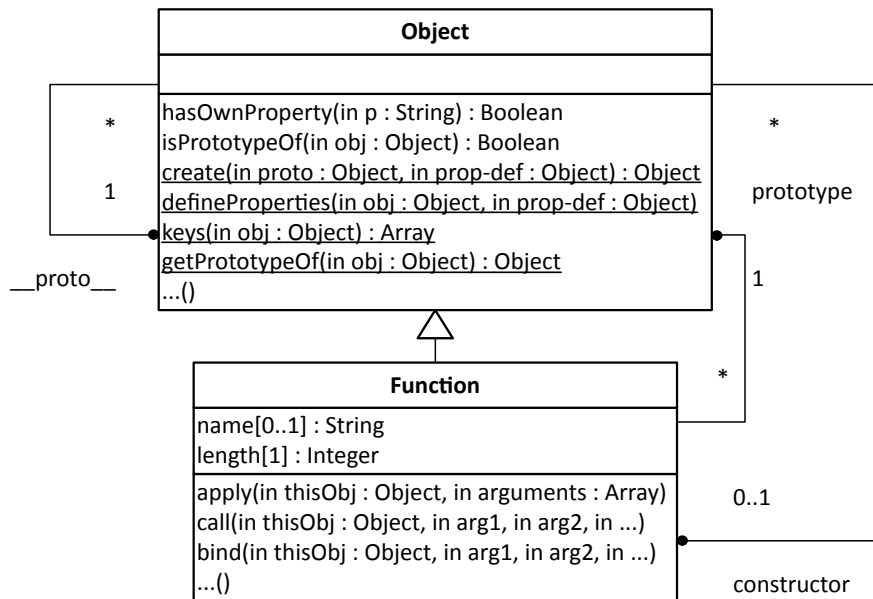
The method `toString` is invoked on the object `pers1` of type `Person` by using the 'dot notation':

```
alert("The full name of the person are: " +
    pers1.toString());
```

When an object `o` is created with `o = new C(...)`, where `C` references a named function with name "C", the type (or class) name of `o` can be retrieved with the introspective expression `o.constructor.name`, which returns "C". However, the `Function::name` property used in this expression is supported by all browsers except Internet Explorer up to version 11.

In JavaScript, a **prototype** is an object with method slots (and sometimes also property slots) that can be inherited by other objects via JavaScript's method/property slot look-up mechanism. This mechanism follows the **prototype chain** defined by the (in ES5 still unofficial) built-in reference property `__proto__` (with a double underscore prefix and suffix) for finding methods or properties. As shown below in Figure 1, every constructor function has a reference to a prototype as the value of its reference property `prototype`. When a new object is created with the help of `new`, its `__proto__` property is set to the constructor's `prototype`. For instance, after creating a new object with `f = new Foo()`, it holds that `Object.getPrototypeOf(f)`, which is the same as `f.__proto__`, is equal to `Foo.prototype`. Consequently, changes to the slots of `Foo.prototype` affect all objects that were created with `new Foo()`. While every object has a `__proto__` property slot (except `Object`), only objects constructed with `new` have a `constructor` property slot.

Figure 1: The built-in JavaScript classes `Object` and `Function`



Notice that we can retrieve the prototype of an object with `Object.getPrototypeOf(o)`, which is an official ES5 alternative to `o.__proto__`

## 10.2 Factory-based classes

In this approach we define a JS object `Person` (actually representing a class) with a special `create` method that invokes the predefined `Object.create` method for creating objects of type `Person`:

```

var Person = {
  typeName: "Person",
  properties: {
    firstName: {range:"NonEmptyString",
      label:"First name",
      writable: true, enumerable: true},
    lastName: {range:"NonEmptyString",
      label:"Last name",
      writable: true, enumerable: true}
  },
  methods: {
    getFullName: function () {
      return this.firstName + " " +
        this.lastName;
    }
  },
  create: function (slots) {
    // create object
    var obj = Object.create( this.methods,
      this.properties);
    // add property for direct type
    Object.defineProperty( obj, "type",
      {value: this, writable: false,
        enumerable: true});
    // initialize object
    Object.keys( slots).forEach(
      function (prop) {
        if (prop in this.properties) {
          obj[prop] = slots[prop];
        }
      }
    );
    return obj;
  }
};
  
```

Notice that the JS object `Person` actually represents a factory-based class. An instance of such a factory-based class is created by invoking its `create` method:

```

var pers1 = Person.create( {firstName:"Tom",
  lastName:"Smith"});
  
```

The method `getFullName` is invoked on the object `pers1` of type `Person` by using the 'dot notation', like in the constructor-based approach:

```
alert("The full name of the person are: " +
    pers1.getFullName());
```

Notice that each property declaration for an object created with `Object.create` has to include the 'descriptors' `writable: true` and `enumerable: true`, as in lines 5 and 7 of the `Person` object definition above.

In a general approach, like in the `mODELcLASSjs` library for model-based development, we would not repeatedly define the `create` method in each class definition, but rather have a generic constructor function for defining factory-based classes. Such a factory class constructor, like `mODELcLASS`, would also provide an *inheritance* mechanism by merging the own properties and methods with the properties and methods of the superclass.

## 11 JavaScript as an Object-Oriented Language

JavaScript is *object-oriented*, but in a different way than classical OO programming languages such as Java and C++. There is no explicit *class* concept in JavaScript. Rather, classes have to be defined in the form of special objects: either as *constructor* functions or as *factory* objects.

However, objects can also be created without instantiating a class, in which case they are *untyped*, and properties as well as methods can be defined for specific objects independently of any class definition. At run time, properties and methods can be added to, or removed from, any object and class. This dynamism of JavaScript allows powerful forms of *meta-programming*, such as defining your own concepts of classes or enumerations.

## 12 The LocalStorage API

For a front-end app, we need to be able to store data persistently on the front-end device. Modern web browsers provide two technologies for this purpose: the simpler one is called *Local Storage*, and the more powerful one is called *IndexDB*.

A Local Storage database is created per browser and per origin, which is defined by the combination of protocol and domain name. For instance, `http://example.com` and `http://www.example.com` are different origins because they have different domain names, while `http://www.example.com` and `https://www.example.com` are different origins because of their different protocols (HTTP versus HTTPS).

The Local Storage database managed by the browser and associated with an app (via its origin) is exposed as the built-in JavaScript object `localStorage` with the methods `getItem`, `setItem`, `removeItem` and `clear`. However, instead of invoking `getItem` and `setItem`, it is more convenient to handle `localStorage` as a map, writing to it by assigning a value to a key as in `localStorage["id"] = 2901465`, and retrieving data by reading the map as in `var id = localStorage["id"]`.

The following example shows how to create an entity table and save its serialization to Local Storage:

```
var persons = {};
persons["2901465"] = {id: 2901465,
    name:"Tom"};
persons["3305579"] = {id: 3305579, name:"Su"};
persons["6492003"] = {id: 6492003,
    name:"Pete"};
try {
    localStorage["personTable"] =
        JSON.stringify( persons);
} catch (e) {
    alert("Error when writing to Local Storage\n"
        + e);
}
```

Notice that we have used the predefined method `JSON.stringify` for serializing the JSON table `persons` into a string that is assigned as the value of the `localStorage` key "personTable". We can retrieve the table with the help of the predefined de-serialization method `JSON.parse` in the following way:

```
var persons = {};
try {
    persons = JSON.parse(
        localStorage["personTable"]);
} catch (e) {
    alert("Error when reading from "+
        "Local Storage\n" + e);
}
```

### **13 Further Reading about JavaScript**

Good open access books about JavaScript are

- [Speaking JavaScript](#), by Dr. Axel Rauschmayer.
- [Eloquent JavaScript](#), by Marijn Haverbeke.
- [Building Front-End Web Apps with Plain JavaScript](#), by Gerd Wagner