JavaScript Summary

Gerd Wagner, 25 Jul 2016

★★★★★ 4.96 (125 votes) Rate this:

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. "In over 20 years programming this is the single best overview of any language ever!"

In over 20 years programming this is the single best overview of any language ever!
[Comment by Member 11032252]

Contents

Introduction

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

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

- implementing advanced user interface components,
- validating user input on the client side,
- automatically pre-filling certain form fields.

dreamed it would become the assembly language for the Web".

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, which is also available as a PDF file, 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 important points of the classical JavaScript summary by Douglas Crockford into consideration.

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, functions, dates and regular expressions are special types 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:

Hide Copy Code

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

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

Hide Copy Code

```
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 added 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 `!!`.

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 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 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 properties and methods inherited from `Object.prototype`. So, a truly empty object has to be created with `null` as prototype, like in `var o = Object.create( null)`.

Table 1: Type testing and conversions

Type	Test if `x` of type	Convert to string	Convert string to type
string	`typeof(x)==="string"`	n.a.	n.a.
boolean	`typeof(x)==="boolean"`	`String(x)`	`Boolean(y)`
(floating point) number	`typeof(x)==="number"`	`String(x)`	`parseFloat(y)`
integer	`Number.isInteger(x)`*)	`String(x)`	`parseInt(y)`
Object	`typeof(x)==="object"`	`x.toString()` or `JSON.stringify(x)`	`JSON.parse(y)`
Array	`Array.isArray(x)`	`x.toString()` or `JSON.stringify(x)`	`y.split()` or `JSON.parse(y)`
Function	`typeof(x)==="function"`	`x.toString()`	`new Function(y)`
Date	`x instanceof Date`	`x.toISOString()`	`new Date(y)`
RegExp	`x instanceof RegExp`	`x.toString()`	`new RegExp(y)`

*) May require a polyfill.

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 code 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

Hide Copy Code

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

Instead, and this is exactly how JavaScript is interpreting this code, we should write:

Hide Copy Code

```
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`.

Strict Mode

Starting from ES5, we can use strict mode for getting more runtime error checking. For instance, 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:

Hide Copy Code

```
'use strict';
```

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

Different Kinds of Objects

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 (hash) map. Objects can be created in an ad-hoc manner, using JavaScript's object literal notation (JSON), without instantiating a class:

Hide Copy Code

```
var person1 = { lastName:"Smith", firstName:"Tom"};
var o1 = Object.create( null);  // an empty object with no slots
```

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.

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):

Hide Copy Code

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

2. Using a map notation:

Hide Copy Code

```
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,

Hide Copy Code

```
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,

Hide Copy Code

```
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 function slots like, for instance,

Hide Copy Code

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

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) architecture paradigm where we have three subnamespaces corresponding to the three parts of an MVC application:

Hide Copy Code

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

A more advanced namespace mechanism can be obtained by using an mmediately 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.

Array Lists

A JS 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 'JavaScript array' creates a terminological ambiguity. But for simplicity, we will sometimes just say 'array' instead of 'JavaScript array'.

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

Hide Copy Code

```
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

Hide Copy Code

```
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`:

Hide Copy Code

```
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:

Hide Copy Code

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

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

Hide Copy Code

```
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:

Hide Copy Code

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

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

Hide Copy Code

```
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 pre-defined array method `forEach`, as in the following example, where the parameter `elem` iteratively assumes each element of the array `a` as its value:

Hide Copy Code

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

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

Hide Copy Code

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

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:

Hide Copy Code

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

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

Hide Copy Code

```
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:

Hide Copy Code

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

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

Hide Copy Code

```
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:

Hide Copy Code

```
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 predefined `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`:

Hide Copy Code

```
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:

Hide Copy Code

```
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:

Hide Copy Code

```
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.

Four Important Types of Data Structures

In summary, the four iportant types of data structures supported by JavaScript 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 2 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 2: 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 in JavaScript is a purely conceptual distinction, which is not supported by the JavaScript execution semantics. For a JavaScript engine, both `{firstName:"Tom", lastName:"Smith"}` and `{"one":1, "two":2, "three":3}` are just objects, and it would interpret the map-style object `{"one":1, "two":2, "three":3}` in the same way as the record-style object `{one:1, two:2, three:3}`. But conceptually, `{firstName:"Tom", lastName:"Smith"}` is a record because `firstName` and `lastName` are intended 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 in a map.

Making such conceptual distinctions helps in the logical deign of a program, and mapping them to syntactic distinctions, even if they are not interpreted differently, helps to better understand the intended computational meaning of the code and therfore improves its readbility.

Procedures and Functions

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 function can be tested with

Hide Copy Code

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

Since JS functions are JS objects, they 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 ***function definition*** is an assignment of a function expression to a variable:

Hide Copy Code

```
var myFunction = function theNameOfMyFunction () {...}
```

where `theNameOfMyFunction` 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 `myFunction()`, and not with `theNameOfMyFunction()`.

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:

Hide Copy Code

```
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 ***function declaration*** has the following form:

Hide Copy Code

```
function theNameOfMyFunction () {...}
```

It is equivalent to the following named function definition:

Hide Copy Code

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

that is, it creates both a function with name `theNameOfMyFunction` and a variable `theNameOfMyFunction` 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:

Hide Copy Code

```
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:

Hide Copy Code

```
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`:

Hide   Copy Code

```
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:

Hide   Copy Code

```
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:

Hide   Copy Code

```
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 artcile JavaScript best practices.

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*** in modern software applications, especially 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*** (with method overriding). 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 (and implemented in the ES6 `class` syntax).

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 and 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 the mODELcLASSjs library, has many advantages, which are summarized in Table 3, the constructor-based approach enjoys the advantage of higher performance object creation.

Table 3. Required and desirable features of JS code patterns for classes

Class feature	Constructor-based approach	Factory-based approach	mODELcLASSjs
Define properties and methods	yes	yes	yes
is-instance-of predicate	yes	yes	yes
direct type property	yes	yes	yes
direct supertype property of classes	no	possibly	yes
Property inheritance	yes	yes	yes
Method inheritance	yes	yes	yes
Multiple inheritance	no	possibly	yes
Multiple classifications	no	possibly	yes
Allow object pools	no	yes	yes

Constructor-based classes in ES6

Only in ES6, a user-friendly syntax for defining constructor-based classes has been introduced (with the new keywords `class`, `constructor`, `static`, `extends` and `super`). This new syntax allows defining a simple class hierarchy in three steps.

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`:

Hide Copy Code

```
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:

Hide Copy Code

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

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

Hide Copy Code

```
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 +")";
  }
}
```

Constructor-based classes in ES5

In ES5, we can define a constructor-based class hierarchy in the form of constructor functions, following a code pattern recommended by Mozilla in their JavaScript Guide. This code pattern requires seven steps for defining a simple class hierarchy. Because such a complex pattern is quite unwieldy, it can be preferable to use a library like cLASSjs for facilitating the definition of constructor-based classes and class hierarchies.

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:

Hide Copy Code

```
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:

Hide Copy Code

```
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

Hide Copy Code

```
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:

Hide Copy Code

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

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

Hide Copy Code

```
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 constructor (`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:

Hide Copy Code

```
// 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:

Hide Copy Code

```
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:

Hide Copy Code

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

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

Hide Copy Code

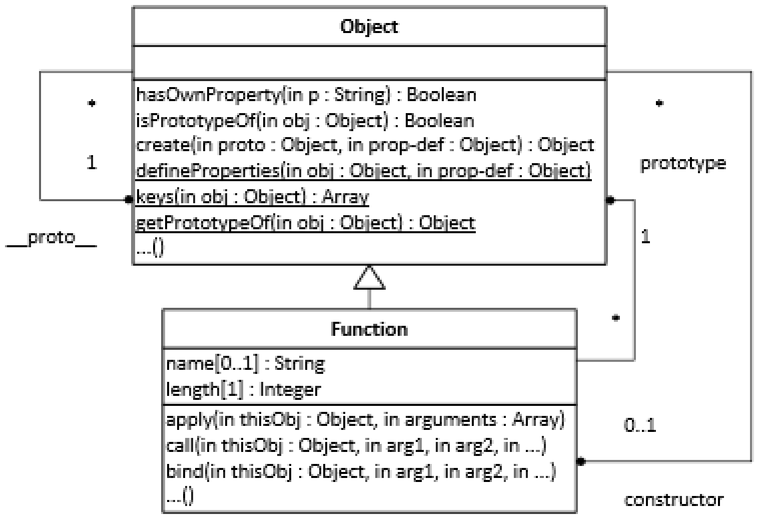
```
alert("The full name of the person are: " +
     pers1.toString());
```

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.

JavaScript's prototype feature

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`.


Object
hasOwnProperty(in p : String) : Boolean
isPrototypeOf(in obj : Object) : Boolean
create(in proto : Object, in prop-def : Object) : Object
defineProperties(in obj : Object, in prop-def : Object)
keys(in obj : Object) : Array
getPrototypeOf(in obj : Object) : Object
...()
Function
name[0..1] : String
length[1] : Integer
apply(in thisObj : Object, in arguments : Array)
call(in thisObj : Object, in arg1, in arg2, in ...)
bind(in thisObj : Object, in arg1, in arg2, in ...)
...()
*
1
__proto__
*
prototype
1
*
0..1
constructor

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`:

Hide Copy Code

```
var Person = {
  name: "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 special property for *direct type* of object
    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:

Hide Copy Code

```
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:

Hide Copy Code

```
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.

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.

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:

Hide Copy Code

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

Notice that we have used the predefined method `JSON.stringify` for serializing the entity table `people` 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:

Hide Copy Code

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

Further Reading

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

History

- July 25, 2016: Improved order of defining/using the term "slot", added table of contents.
- May 4, 2016: Added a new section on the localStorage API.
- March 7, 2016: Improved explanation of data structures, added topic "method overriding", a hyperlink and additional subheadings to section on classes.
- January 15, 2016: New hyperlink added; improved comparison table and explanations in the section "Defining and using classes".
- November 9, 2015: Added a paragraph about the string datatype; added paragraphs about "arguments", call/apply and bind in the section about JS functions; added sidebar about ES6 classes, improved discussion of constructor-based classes.
- October 21, 2015: Added Date and RegExp, added Table 1 "Type testing and conversions", corrected links.

- September 20, 2015: Added ES 2015+2016 hint, parseFloat, Function::name, Function::length, example for entity tables, paragraph about cloning maps.
- August 31, 2015, added requirements for class code patterns and explanation of subtyping with constructor-based classes, and a diagram.
- July 16, 2015, added further explanations about defining and using classes in JS
- July 7, 2015, added 1) link to Crockford's classcial survey, 2) remarks about isInteger and about number-to-string conversion. 3) link to CP article on mODELcLASSjs
- July 2, 2015, first version
- January 15, 2016: New hyperlink added; improved comparison table and explanations in the section "Defining and using classes".
- November 9, 2015: Added a paragraph about the string datatype; added paragraphs about "arguments", call/apply and bind in the section about JS functions; added sidebar about ES6 classes, improved discussion of constructor-based classes.
- October 21, 2015: Added Date and RegExp, added Table 1 "Type testing and conversions", corrected links.

- September 20, 2015: Added ES 2015+2016 hint, parseFloat, Function::name, Function::length, example for entity tables, paragraph about cloning maps.
- August 31, 2015, added requirements for class code patterns and explanation of subtyping with constructor-based classes, and a diagram.
- July 16, 2015, added further explanations about defining and using classes in JS
- July 7, 2015, added 1) link to Crockford's classcial survey, 2) remarks about isInteger and about number-to-string conversion. 3) link to CP article on mODELcLASSjs
- July 2, 2015, first version

License

Share

About the Author

Gerd Wagner

Instructor / Trainer
Germany

Researcher, developer, instructor and cat lover.
Co-Founder of http://web-engineering.info

You may also be interested in...

Smart Validation Summary

Modal Popup From Scratch

Custom Validator and Validation Summary

What is SCRUM?

Tetris on Canvas

ArrayList Sort Tutorial

Comments and Discussions

You must Sign In to use this message board.

Search Comments

First Prev Next

Ah , Thanks
VernonM 9-Nov-16 5:43

Ah , Thanks

Sign In · View Thread

records v maps
VernonM 8-Nov-16 7:49

Hello & Thanks ,
Sorry but , I am not getting it .
What is the difference between :

Hide Copy Code

```
records, which are special JS objects, such as {firstName:"Tom", lastName:"Smith"}, as discussed
above,

maps, which are also special JS objects, such as {"one":1, "two":2, "three":3}, as discussed above,
```

Thanks

Sign In · View Thread 5.00/5 (1 vote)

Re: records v maps
Gerd Wagner 8-Nov-16 9:35

A JS object like {firstName:"Tom", lastName:"Smith"} represents a record corresponding to what is called a "struct" in C/C++/C#. For instance, we could define a record structure like

Hide Copy Code

```
struct person_record {
  public string firstName;
  public string lastName;
};
```

and then instantiate it with

Hide Copy Code

```
person_record pers1;
pers1.firstName = "Tom";
pers1.lastName = "Smith";
```

In many use cases, we have to deal with many records of the same type/structure, forming a table.

This does not hold for maps, which do not have a fixed structure (or number of entries) and represent rather sets (of key-value pairs).

Sign In · View Thread

Record vs map
Member 12061600 21-Oct-16 4:24

I can't see where you define differences between map and a record?

also why

return ((x[0] === y[0]) ? x[1]-y[1] : x[0]-y[0]);

[1] above? and why indexes anyway instead of just x and y?

modified 21-Oct-16 10:12am.

Sign In · View Thread

Re: Record vs map
Gerd Wagner 23-Oct-16 2:56

Records and maps are explained in the section "Different Kinds of Objects".

Records are JS objects that only have property-value slots, like for instance,

Hide Copy Code

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

where `firstName`, `lastName` and `age` are intended to denote properties (or 'record fields').

Maps are JS objects that only have key-value slots for looking up values based on their keys like, for instance,

Hide Copy Code

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

where "one", "two" and "three" are not intended to denote properties/fields, but are just arbitrary string values used as keys in a map.

This is a purely conceptual distinction, which is not supported by the JavaScript execution semantics. For a JavaScript engine, records and maps are just objects.

The comparison function that has to be provided as an argument when invoking the array `sort` method, needs to compare two elements of the given array. Since this example is intended to sort an array of two-element arrays (representing points in a 2D space), we need to compare the elements of the inner two-element arrays.

The conditional expression in the return statement tests if the first elements (x[0] and y[0]) are the same. If they are the same, then the second elements have to be compared with each other: if x[1]-y[1] is greater than 0, then we obtain x > y; if its value is 0, we get x === y, and if it's smaller than 0, x < y.

Sign In · View Thread

Re: Record vs map
Member 12061600 9-Dec-17 11:57

records, which are special JS objects, such as {firstName:"Tom", lastName:"Smith"}, as discussed above,

maps, which are also special JS objects, such as {"one":1, "two":2, "three":3}, as discussed above,

This distinction was really not necessary IMHO as you also note in article they are same.
But OK.

modified 9-Dec-17 17:05pm.

Sign In · View Thread

My vote of 5
marekola.bis 27-Jul-16 9:44

Just awesome!

Sign In · View Thread

Thank you!
MarcusCole6833 25-Jul-16 7:58

A lucid piece, and I finally got to understand how to use Prototype!

Sign In · View Thread

5ed
Karthik_Mahalingam 30-Jun-16 20:30

Good information provided

Sign In · View Thread

My vote of 5
Member 11358445 5-May-16 12:59

Fantastic summary

Sign In · View Thread

Not clear about a code example
Shanmuga Sundaram R 19-Apr-16 21:03

Hi
Well written and easy to understand article. Thanks.

I have a question. In a code example (given below), list.sort sorts the list. What's the purpose of the function within list.sort?

Hide Copy Code

```
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]);
});
```

Sign In · View Thread

Re: Not clear about a code example
Gerd Wagner 20-Apr-16 7:46

The predefined JS function sort requires a comparison function as its parameter. This function must return a positive number if x>y, 0 if x=y, and a negative number if x<y.

For comparing two pairs of integers like (3,2) and (2,3), we use the lexicographic ordering: first compare their first components, and then their second. In my example, I'm providing an anonymous function that (a) returns a positive number if x[0] > y[0] or if x[0] = y[0] and x[1] > y[1], (b) returns 0 if x[0] = y[0] and x[1] = y[1], (c) returns a negative number if x[0] < y[0] or if x[0] = y[0] and x[1] < y[1].

Sign In · View Thread

Re: Not clear about a code example
Shanmuga Sundaram R 20-Apr-16 21:03

The code seemed to work with just list.sort, without the lexicographic comparison function, with the sample data provided.

When I changed the data to

Hide Copy Code

```
[10,2],[1,3],[1,1],[2,1]
```

the default sort failed because it was using Unicode based comparison. Your explanation helped me understand this nuance better.

Thanks again.

Sign In · View Thread

Good Material

'Mualle Mats'ela 13-Apr-16 3:49

Very useful summary!

Sign In · View Thread

Summary of summary

Prefabrykaty 10-Mar-16 3:52

Great summary!

Sign In · View Thread

Nice Summary

Alireza_1362 8-Mar-16 6:40

Thanks

Sign In · View Thread 5.00/5 (1 vote)

Real Time-liness

anaQata 7-Feb-16 11:02

I am really grateful for your amazing article. It really gives a solid foundation for anyone delving into Javascript.
However, I would suggest that in future, you may add up some real time examples of where javascript is/may be used in a real time website.
Lets say the topic on arrays... a useful way might be to present instances where you might require to use javascript arrays in a real website.
In either case, I think this is the best article i've found that comprehensively and acutely summarizes javascript.
what a 5-star looks and feels like

Sign In · View Thread

Re: Real Time-liness

Gerd Wagner 7-Feb-16 12:03

Thanks for your positive feedback. With "real time examples", do you refer to JavaScript techniques/APIs (such as WebSockets) for real-time communication? Notice that I've recently co-authored an article on the Web Real-Time Com munication API WebRTC.

Sign In · View Thread

Re: Real Time-liness

anaQata 9-Feb-16 10:02

Thanks for you prompt reply sir. I mean the specific areas that these javascript topics apply. In a topic like looping, it

becomes hard as a newbie to comprehend where, in front end design, such a concept might apply. My suggestion was for you to write the article with a few sample websites/ scenarios where the applicability of such concepts may or already apply.
Example: Arrays are useful in form validation as they can be used to store data useful in validation (can't tell if my statement is right).
But i guess you get my point...

Sign In · View Thread

My vote of 5
zpaulo_carraca 19-Jan-16 1:42

Thanks, doc.

Sign In · View Thread

My vote of 5
arroway 9-Nov-15 22:20

Very good article

Sign In · View Thread

My vote of 5
Dan Moyer 30-Oct-15 5:35

Excellent overview. Good references at end of where to get more detail.

Sign In · View Thread

Useful info
Murali Vijay 9-Oct-15 3:23

i just read the document .it was very useful information.keep posting

Sign In · View Thread

Brilliant
Member 11032252 3-Sep-15 21:35

In over 20 years programming this is the single best overview of any language ever!

Sign In · View Thread

Re: Brilliant
Gerd Wagner 3-Sep-15 22:36

Thanks for the nice compliment!

Sign In · View Thread

Refresh 1 2 Next »

General News Suggestion Question Bug Answer Joke Praise Rant Admin