Tecnologie e applicazioni web

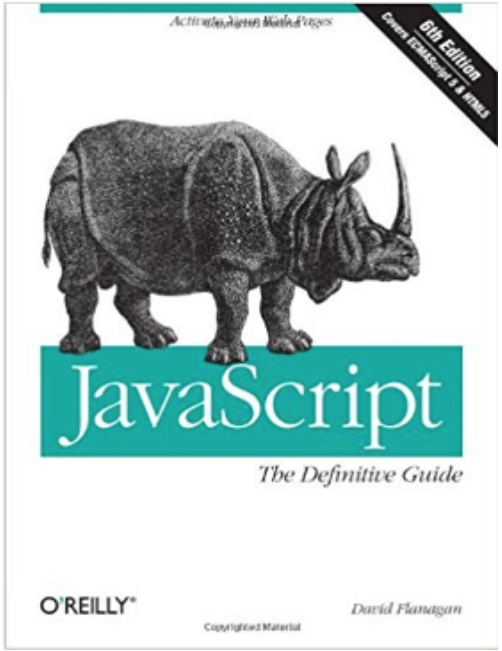
JavaScript

Filippo Bergamasco (filippo.bergamasco@unive.it)
http://www.dais.unive.it/~bergamasco/
DAIS - Università Ca'Foscari di Venezia
Academic year: 2021/2022

Atwood's Law: any application that can be written in JavaScript will eventually be written in JavaScript.

—Jeff Atwood

Suggested readings



Introduction

JavaScript is the most common dialect (implementation) of a language standardized by Netscape to the European Computer Manufacturer's Association known as **ECMAScript**.

Almost all browsers support ECMAScript version >5

Introduction

Despite its standard name, in practice just about everyone calls the language “JavaScript”.

It is a **high-level**, **dynamic**, **untyped**, **interpreted** programming language. It is well-suited to object-oriented and functional programming styles.

Introduction

The core JavaScript language defines a minimal API for working with text, arrays, dates and regular expressions but non include any input-output functionality.

Input and output are delegated to the “host environment” in which the JavaScript is embedded. This can be the browser or a stand-alone environment like Node.js

JavaScript Pillars

1. **Prototype inheritance**
   a. OOP not «class based»
   b. OLOO: objects linked to other objects

2. **Functional programming**
   a. Lambda / anonymous functions
   b. Closures

Data types

JavaScript types can be divided in two categories:

1. **Primitive** types:
    - Numbers
    - Strings (text)
    - Booleans (true/false)
    - null and undefined

Data types

JavaScript types can be divided in two categories

2. **Object** types: (everything that is not primitive)
    - An object is a collection of **properties**, where each property has a **name** and a **value**.
    - An array is an ordered collection of numbered values
    - A function, that is an object with executable code associated to it

Data types

Types can be categorized as **mutable** and **immutable**. (A mutable type can change its value, an immutable type can not).

- **Objects** and arrays are **mutable**.
- **Primitive** types are **immutable** (also strings are immutable even if you can manipulate it)

null / undefined

null and undefined are two types used to indicate the absence of a value. What's the difference between the two?

- **undefined** represent a system-level, unexpected or error-like absence of a value
- **null** represent a program-level, normal or expected absence of a value

Objects

An object is an unordered collection of **properties**, each of which has a **name** and a **value**.

Property names are always **strings.**

Other than its properties, an object inherits properties from another object known as its "**prototype**"

Objects are **mutable** and manipulated by **reference**.

Objects

Object's properties can be accessed using the dot `.` or `[ ]` operators.

```
User.name = "Filippo"
User["name"] = "Filippo"
```

What's the difference?

Objects

```
User.name = “Filippo”
```

- “name” must be known a-priori, therefore this expression implies a finite number of properties

```
User[“name”] = “Filippo”
```

- Objects can be used as associative arrays, where the property name is any expression that can be converted to a string. Object properties are not known a-priori.

The global object

In JavaScript there exists a special object named “the global object” containing the globally defined symbols that are available to the JavaScript program.

Such object depends by the context:

- Browser: global object is a **window**
- Worker: global object is a **WorkerGlobalScope**
- Node.js: global object is called **global**

Objects

An object can be created in 3 different ways:

1. Using object literals

```
var book = { “title”:”Javascript”, pages:200 }
```

2. Using the **new** keyword, followed by a function invocation. Such function is named «constructor»

```
var a = new Array(); var b = new Date();
```

3. By invoking Object.create(<prototype>)

```
var o = Object.create( {x:10, y:20} )
```

Prototype

- `Object.prototype` is the prototype of all the objects created with an object literal expression.
- Objects created with the **new** keyword will use the prototype of the constructor (note: not the constructor itself!)

Ex: var a = new Array();

Will have Array.prototype as prototype

.

Prototype

In JavaScript any object **o** contains a reference to another object (or `null`, in some cases) named **prototype**.

The object o inherits all the properties of its prototype. When we access a property of an object, the «prototype chain» is used to lookup the property value.

Prototype

When we **read** an object property, the prototype chain of that obect is followed until the property is found (or null is encountered).

When we **write** a property to o:

- If the property is read-only, the assignment is not allowed
- If the property is inherited, the assignment creates a new property in o without modifying any object in the prototype chain. (override)

Functions

Functions are the fundamental building blocks of all JavaScript applications:

- JavaScript is said to have first-class functions
- Functions are objects!
- Functions can be defined at runtime
- Closures are supported

Function definition

Two ways to define a function:

- Function declaration

```
function sum(a,b) { return a+b; }
```

- Function expression

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

…or using arrow functions (ES6):

```
var sum = (a,b) => { return a+b; }
```

Function definition

Function declaration:

- Conditional declaration is not allowed in some interpreters
- Function is **Hoisted**
- Function will automatically have a name (useful for debugging)

Function definition

Function expression:

- Conditional declaration allowed
- Function is not **Hoisted**.
- Function is anonymous, and can be accessed only through the variable identifier to which it has been assigned.

Scope and Hoisting

Function scope:

All the variables are visible in the function code block in which are contained (and in all the nested functions)

Hoisting:

Everything (variables and functions) declared inside a function is visible everywhere inside that function (like if it has been declared at the beginning)

Invoking functions

A function can be invoked in 4 different ways:

- As an invocation expression: `f(a)`
- As a method invocation (if the function is a property of an object) :

  `o.f(a) ; o["f"](a)`
- As a constructor invocation if preceded by the new keyword: `var o = new Object()`
- Indirectly using `call()` or `apply()`

this

This keyword references the execution context of a function.

- Outside a function definition block, **`this`** refers to the global object
- Inside a function invoked as expression:
  - **`this`** is undefined if we are in strict mode, and refers to the global object otherwise

this

This keyword references the execution context of a function.

- Inside a function invoked as a method, **`this`** refers to the object on which the function is invoked.
  - NOTE: Doesn't matter where the function is defined, only how is invoked!

this

This keyword references the execution context of a function.

- In an object prototype chain, this always refers to the object on which a method is called, not the object that actually contains that method
  - This is an interesting fact of the prototype-based inheritance of JavaScript

this

This keyword references the execution context of a function.

- If the function is invoked as a constructor, **this** refers to the newly created object.
- If the function inherits from `Function.prototype` and is invoked using `call()` or `apply()`, `this` refers to the object passed as first argument of `call()` or `apply()`.

.

Closures

JavaScript uses a so-called «lexical scoping»: functions are executed using the variable scope according to **when they have been defined**, and not the scope existing **when they are invoked**.

Combination of a function and the scope in which variables are resolved is called closure (chiusura)

Closures

Usually functions are defined and invoked in the same scope, so closures are not visible.

Interesting things happen when a function is invoked with a different invokation scope…

Closures

```
var scope = "global scope";
function checkscope() {
      var scope = "local scope";
      function f() { return scope; }
      return f();
}
checkscope() // what is the return value?
```

Closures

```
var scope = "global scope";
function checkscope() {
      var scope = "local scope";
      function f() { return scope; }
      return f;
}
checkscope()() // what is the return value?
```

Closures

```
var scope = "global scope";
function checkscope() {
      var scope = "local scope";
      function f() { return scope; }
      return f;
}
checkscope()() // what is the return value?
```

var scope=”local scope” still exists after the checkscope() definition (even if it’s local for the function!) because remains **bound** to the scope of f()

Closures

- Nested functions can access variables of the outer function even if the outer function returns and hence does not exists anymore.

- Nested functions keep **references** (not the values) to variables of the outer function. Useful to use functions to simulate private variables.

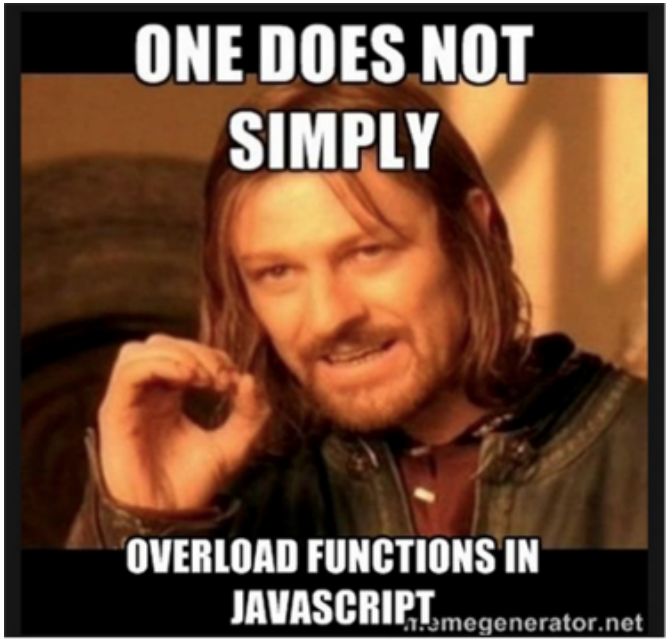
Arguments

JavaScript functions accept a variable number of parameters even if not explicitly set in the function signature (variadic functions).

- If less parameters are passed, the missing ones will be set to `undefined`
- If more parameters are passed, the `arguments` object can be used to retrieve their values

Function overloading

JavaScript does not support functions with the same name but different parameter list. The last one defined overrides the former.



Classes

OOP is based to the concept of **classes** of objects sharing the same set of properties.

In JavaScript < ES6 you cannot explicitly define classes. However, we can simulate classes by using the prototype chain:

If two objects inherit properties from the same prototype object we say that such objects are instances of the same class.

Constructors

The idiomatic way to instantiate objects of a class is by using the constructor.

When using the keyword **new**, constructor's prototype is used to set the prototype of the newly created object.

- All objects created with the same constructor will have the same prototype. Therefore, all such objects will be instances of the same class

Constructors

NOTE: The «identity» of a class is defined by the constructor's prototype, not the actual constructor used to instantiate class elements:

- Two different constructors may have the same prototype and therefore will produce elements of the same class.

The `instanceof` operator allow us to determine if two objects belong to the same class.

Classes

Classes can be dynamically extended by modifying the prototype.

This is possible even after the object has been instantiated!

This allow us to «inject» new features to a pool of objects.
For example, we can modify the behaviour of built-in objects like the strings

.

Subclasses

A class B extends a class A if it inherits all its attributes (and methods). Moreover, B can contain additional attributes and methods or even overwrite the existing ones (override).

We can simulate subclasses by carefully initializing the subclass prototype

Subclasses

If B extends A then B.prototype must be a child (ie. descendent in the hierarchy) of A.prototype. This way, all properties in A will be automatically available in B (thanks to the prototype chain).

`call()` or `apply()` can be used to forcefully invoke superclass methods (like the super keyword used in other programming languages).

.

Composition

Class composition is conceptually easier to implement:

- Superclass is included as an attribute of the «subclass»
- Additional functions are implemented by forwarded all the inherited methods

Composition is usually better than classical inheritance. Why?

Problems with classical inheritance

Tight coupling

Subclasses tightly depend on superclasses, usually knowing and accessing the internal details

Non-flexible hierarchies

Classical inheritance usually creates a static «taxonomy» that rarely fits well to the problem. The result is that code is often duplicated between super- and sub-classes

Problems with classical inheritance

Multiple inheritance

Can be complex to implement and usually inconsistent with respect to single inheritance

Gorilla/Banana problem

Sometimes there are functions in the superclass that you don't want to inherit (because they make no sense in the subclass). Classical inheritance does not allow you to select what to inherit

Mixins

Mixins is a technique in which we combine the advantages of working with interfaces with the code reuse given by class inheritance

Objects are «containers of functions» that can be borrowed and mixed to create other objects.

.

Type coercion

('b'+'a'++'a'+'a).toLowerCase() -> "banana"

Or, taken to the extreme:

https://www.youtube.com/watch?v=sRWE5tnaxlI

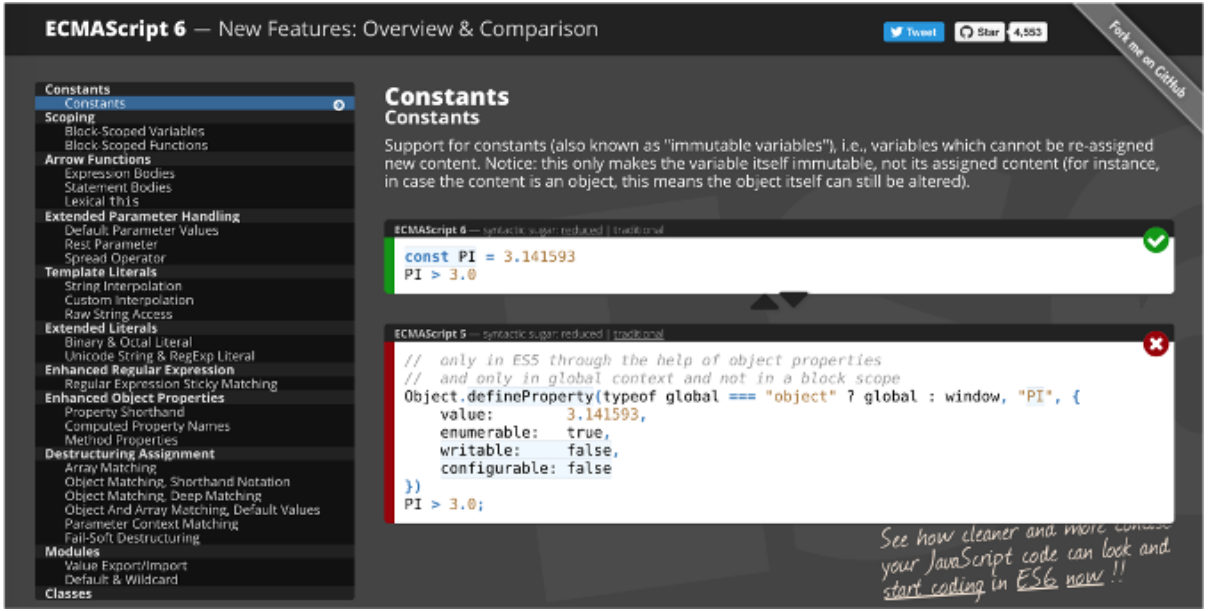
Strict mode

- Strict mode can be used to reduce the most common errors caused by the JavaScript syntax
- Can be applied to an entire script or to a single function:

```
function test () {
 'use strict';
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

ES 6 – Important features



http://es6-features.org/

ES 6 – Arrow functions

Operator **=>** (arrow function) can be used to create anonymous functions without using the **function** keyword:

```
( <args> ) => <expression/function body>
```

Arrow functions support the so-called «lexical this». **this** used inside an arrow function points to to the same object as the object pointed by the **this** keyword on the contained object.

Classes

ES6 introduces special functions used to simplify the object creation an manage classical inheritance:

```
class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }

  get area() { return this.calcArea() }

  calcArea() { return this.height * this.width; }
}
```