Tecnologie e applicazioni web

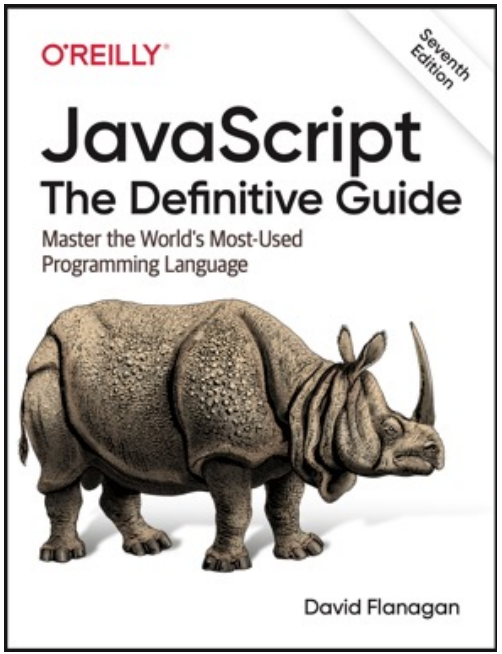
JavaScript

Filippo Bergamasco (filippo.bergamasco@unive.it)
http://www.dais.unive.it/~bergamasco/
DAIS - Università Ca'Foscari di Venezia
Academic year: 2023/2024

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 **E**uropean **C**omputer **M**anufacturer's **A**ssociation known as **ECMAScript**.

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**, **interpreted**, **untyped**, and **dynamic** programming language well-suited to object-oriented and functional programming styles.

Introduction

The core JavaScript language defines minimal APIs for working with text, arrays, dates, and regular expressions but does not include 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 into 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 cannot).

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

null / undefined

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

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

Variable declaration

In modern JavaScript, variables are declared with the `let` keyword.

- If an initial value is not specified, the value is **`undefined`** until the first assignment
- Multiple declarations can be stacked:
  - `let i,j,k`
- A JavaScript variable can hold values of any type (no type check performed). `let a=10; a="ten";`

The `const` keyword is used for constants.

Variable scope

Variables and constants declared with let/const are **block-scoped**.

Variables declared as part of a for loop have the loop body as their scope, even though they technically appear outside the curly braces.

Notes:

- It is a syntax error to use the same name with more than one let or const declaration in the same scope.
- It is legal to declare a new variable with the same name in a nested scope.

Global variables

When a declaration appears at the top level (outside any code block), the variable is said to be a **global variable**.

- In Node.js the scope of a global variable is the file that it is defined in.
- In the browser, the scope of a global variable is the HTML document (i.e. the page) in which it is defined

Variables declared with var

Variables can also be declared with the `var` keyword:

- There is no block scope. They are scoped to the body of the containing function no matter how deeply nested are inside that function
- It is legal to declare the same variable multiple times
- var declarations are **hoisted** (ie. the declaration is lifted up to the top of the enclosing function, while the initialization remains where you wrote it)

Variables declared with var

- Global variables declared with `var` are implemented as properties of the global object. They cannot be deleted from the **delete** operator.
- Global variables declared with let and const are not properties of the global object.

Objects

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

A property name can be a **string** or a **Symbol().**

- Other than its properties, an object inherits properties from another object known as its "**prototype**": OLOO
- Objects are **mutable** and manipulated by **reference**.

Property access

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

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

What's the difference?

Property access

`user.name = “Filippo”`

- “name” must be a legal identifier (known apriori). 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 apriori.

Property access

```
user.name = “Filippo”
user[“name”] = “Filippo”
```

With either type of property access expression, the expression before the . or [] is first evaluated. If the value is null or undefined, the expression throws a **TypeError**, since these are the two JavaScript values that cannot have properties

- Since ES2020 conditional property access is possible:

```
let username = user?.name
```

The global object

In JavaScript, a special object named “the global object” contains the globally defined symbols available to the JavaScript program.

Such object depends on 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

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

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

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

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

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

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 look up the property value.

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: `let a = new Array();`

`a` will have Array.prototype as prototype

.

`__proto__` vs. `prototype`

Note:

- the prototype of any object is accessible through the property `__proto__`, but is not meant to be modified directly
- Every function have a property (is an object!) named `prototype` that will be used as “blueprint” to set the `__proto__` property of the newly created object when the `new` keyword is used

Prototype

When we **read** an object property, the object's prototype chain 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)

Deleting properties

The delete operator can be used to remove a property from an object:

```
delete book.author
```

Note: Inherited properties must be deleted from the prototype object in which they are defined

Testing and enumerating properties

The `in` operator evaluates to true if an object has a specific property:

```
let o = {x:1, z:3};
“x” in o; //true
“y” in o; //false
```

Since properties are enumerable by default, you can iterate through them with:

```
for (let p in o ) { console.log(p); // x z }
```

Spread operator

The spread operator **...** can be used to copy properties of an existing object into a new object:

```
let position={x:0,y:0};
let dimensions = { width: 100, height: 75 };
let rect = { ...position, ...dimensions };
rect.x + rect.y + rect.width + rect.height
```

Note: available only inside object literals!

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 anonymous
- 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

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

- Arrow functions (since ES6):

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

Function declaration

- Is a statement
- Declares a variable and assigns a function object to it. The name of the function is the name of the newly created variable.
- Conditional declaration is not allowed (in some interpreters and strict mode)
- Function is **Hoisted**
- Function will automatically have a `name` property (useful for debugging)

Function expression

- Is an expression
- Does not declare a variable: it is up to you to assign the newly defined function object to a constant or variable if you are going to need to refer to it
- 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.

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: `let o = new Object()`
- Indirectly, using `call() or apply()`

Conditional invocation

Since ES2020 you can also use:

```
f?.(x)
```

Which is equivalent to:

```
(f !== null && f !== undefined) ? f(x) : undefined
```

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: It doesn't matter where the function is defined, only how it 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 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 the first argument of `call()` or `apply()`.

.

Arrow functions

- They do not have a prototype property, so they cannot be used as constructors.
- The `this` keyword is inherited from the environment in which they are defined rather than when they are invoked: **lexical this**.

Variadic functions

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

Closures

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

The combination of a function and the scope in which variables are resolved is called Closure

Closures

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

Exciting things happen when a function is invoked with a different invocation scope…

Closures

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

Closures

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

Closures

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

`let 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 exist anymore.

- Nested functions keep **references** (not the values) to the outer function's variables making it useful to simulate private variables.

Classes

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

In JavaScript, classes use prototype-based inheritance:

If two objects inherit properties from the same prototype object, we say 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 allows 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 allows us to «inject» new features into a pool of objects. For example, we can modify the behavior of built-in objects like the strings

.

Subclasses

In OOP we say that 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).

.

Class keyword

Starting from ES6, classes can be created with the `class` keyword.

Note: The `class` keyword does not alter the fundamental nature of JavaScript's prototype-based classes.

It is essentially a syntactic sugar that is tied to the mechanisms we have seen so far.

Class keyword

- Classes can also be defined with expressions:

```
let square = class { constructor(x) { this.area = x*x:}};
```

- All code within the body of a class declaration is implicitly in strict mode
- Unlike function declarations, class declarations are not hoisted: a class cannot be instantiated before being declared.