**Speaking JavaScript**

Axel Rauschmayer

**Revision Chapters**

**Chapter 1**

Syntax > Statements Versus Expressions >

|  |
| --- |
| myFunction(y >= 0 ? y : -y) |

**Chapter 9**

The void Operator (**revision**)

**Categorizing Values via typeof and instanceof**

Pitfall: typeof null

**Chapter 13**

Labels

Chapter 15

self-recursion function

Chapter Summary:

# Chapter 1: Basic JavaScript

1. Basic JavaScript

Background

Syntax

Variables and Assignment

Values

Booleans

Numbers

Operators

Strings

Statements

Functions

Exception Handling

Strict Mode

Variable Scoping and Closures

Objects and Constructors

Arrays

Regular Expressions

Math

Other Functionality of the Standard Library

This chapter is about “Basic JavaScript,”

You don’t have to learn everything at once, which can be confusing.

**\*\*\* Background**

**++ JavaScript Versus ECMAScript:**

ECMAScript is the official name for JavaScript.

\* JavaScript means the programming language.

\* ECMAScript is the name used by the language specification.

**+++ Influences and Nature of the Language**

JavaScript’s creator, Brendan Eich, borrowed from several programming languages: Java (syntax, primitive values versus objects), Scheme and AWK (first-class functions), Self (prototypal inheritance), and Perl and Python (strings, arrays, and regular expressions).

**\*\*\* Syntax**

This section explains basic syntactic principles of JavaScript.

**+++ An Overview of the Syntax**

|  |
| --- |
| // two slashes start single-line comments |

|  |
| --- |
| **var** x; // declaring a variable  x = 3 + y; // assigning a value to the variable `x`  foo(x, y); // calling function `foo` with parameters `x` and `y` obj.bar(3); // calling method `bar` of object `obj` |

// A conditional statement

|  |
| --- |
| **if** (x === 0) { // Is `x` equal to zero? x = 123; } |

// Defining function `baz` with parameters `a` and `b`

|  |
| --- |
| **function** **baz**(a, b) { **return** a + b; } |

Note the two different use of the equal sign:

\* A single equal sign (=) is used to assign a value to a variable.

\* A triple equal sign (===) is used to compare two values

**+++ Statements Versus Expressions**

\* Statements “do things.”. A program is a sequence of statements.

Example:

|  |
| --- |
| **var** foo; |

\* Expressions produce values. They are function arguments, the right side of an assignment, etc.

Example: 3 \* 7;

JavaScript has two different ways to do if-then-else ---

either as a statement:

|  |
| --- |
| **var** x; **if**(y >= 0) { x = y; } **else** { x = -y; } |

or a expression:

|  |
| --- |
| **var** x = y >= 0 ? y : -y; |

**+++ Semicolons**

Semicolons are optional in JavaScript. But recommend always including them.

**+++ Comments**

JavaScript has two kinds of comments: single-line comments and multiline comments.

Single-line comments start with // and are terminated by the end of the line:

|  |
| --- |
| x++; // single-line comment |

Multiline comments are delimited by /\* and \*/:

|  |
| --- |
| /\* This is  a multiline comment. \*/ |

**\*\*\* Variables and Assignment**

Variables in JavaScript are declared before they are used:

|  |
| --- |
| **var** foo; // declare variable `foo` |

**+++ Assignment**

You can declare a variable and assign a value at the same time:

|  |
| --- |
| **var** foo = 6; |

you can also assign a value to an existing variable:

|  |
| --- |
| foo = 4 // change variable `foo` |

**+++ Compound Assignment Operators**

There are compound assignment operators such as +=. The following two assignments are equivalent:

|  |
| --- |
| x += 1; x = x + 1; |

**+++ Identifiers and Variable Names**

Identifiers are case sensitive.

The first character of an identifier can be any Unicode letter, a dollar sign ($), or an underscore (\_). Subsequent characters can additionally be any Unicode digit. Thus, the following are all legal identifiers:

|  |
| --- |
| arg0 \_tmp $elem n |

Reserved words cannot be used as variable names (including function names & parameter names):

--- all JS syntax keywords. See MDN ---

The following three identifiers are not reserved words, but you should treat them as if they were:

|  |
| --- |
| **Infinity** **NaN** **undefined** |

Lastly, you should also stay away from the names of standard global variables (see Chapter 23). You can use them for local variables without breaking anything, but your code still becomes confusing.

**\*\*\* Values**

JS has many values that we have come to expect from programming languages: booleans, numbers, strings, arrays, and so on. All values in JS have properties.

Each property has a key (or name) and a value. You can think of properties like fields of a record. You use the dot (.) operator to read a property:

value.propertyKey

For example, the string ‘abc’ has the property length:

|  |
| --- |
| **var** str = 'abc'; str.length; // 3 |

The preceding can also be written as:

|  |
| --- |
| 'abc'.length // 3 |

The dot operator is also used to assign a value to a property:

|  |
| --- |
| **var** obj = {}; // empty object obj.foo = 123; // create property `foo`, set it to 123 obj.foo; // 123 |

And you can use it to invoke methods:

‘hello’.toUpperCase(); // HELLO

In the preceding example, we have invoked the method toUpperCase() on the value ‘hello’.

**+++ Primitive Values Versus Objects**

\* The primitive values are booleans, numbers, strings, null, and undefined.

\* All other values are objects.

A major difference between the two is how they are compared; each object has a unique identity and is only (strictly) equal to itself:

|  |
| --- |
| **var** obj1 = {}; // an empty object **var** obj2 = {}; // another empty object obj1 === obj2 // false obj1 === obj1 // true |

In contrast, all primitive values encoding the same value are considered the same:

|  |
| --- |
| **var** prim1 = 123; **var** prim2 = 123; prim1 === prim2; // true |

**+++ Primitive Values**

- Booleans: true, false

- Numbers: 1736, 1.321

- Strings: ‘abc’, “abc”

- Two 'non values’: undefined, null

Compared by value

the content is compared

|  |
| --- |
| 3 === 3 // true 'abc' === 'abc' // true |

Always immutable

Properties cannot be changed, added, or removed:

|  |
| --- |
| **var** str = 'abc'; str.length = 1; // no effect str.length; // 3 |

|  |
| --- |
| str.foo = 3; // try to create property `foo` str.foo; // undefined / no effect, unknown property |

(Reading an unknown property always returns undefined)

**+++ Objects**

All nonprimitive values are objects.

**\* Plain object,** which can be create by object literals

|  |
| --- |
| { firstName: 'Jane', lastName: 'Doe' } |

The preceding object has two properties: the value of property firstName is ‘Jane’ and the value of property lastName is ‘Doe’.

**\* Array,** which can be create by array literals

|  |
| --- |
| ['apple', 'banana', 'cherry'] |

The preceding array has three elements that can be accessed via numeric indices. For example, the index of ‘apple’ is 0.

**\* Regular expressions,** which can be created by regular expression literals

|  |
| --- |
| /^a+b+$/ |

-Objects have the following characteristics:

-Comparedby reference

-Identities are compared; every value has its own identity:

|  |
| --- |
| {} === {} // false (two different empty objects) |

|  |
| --- |
| **var** obj1 = {}; // an empty object **var** obj2 = {}; // another empty object obj1 === obj2 // **false** obj1 === obj1 // **true** |

-Mutable by default

You can normally freely change, add, and remove properties

|  |
| --- |
| **var obj = {}; obj.foo = 123; // add property `foo` obj.foo; // 123** |

**+++ undefined and null**

Most programming languages have values denoting missing information. JavaScript has two such ‘non value’, **undefined** and **null:**

\* undefined means ‘no value’. Uninitialized variables are undefined:

|  |
| --- |
| **var foo; foo // undefined** |

\* Missing parameters are undefined:

|  |
| --- |
| **function f(x) { return x} f(); // undefined** |

\* If you read nonexistent property, you get **undefined**:

|  |
| --- |
| **var obj = {}; obj.foo // undefined** |

\* **null** means ‘no object.’ It is used as a nonvalue whenever an object is expected (parameters, last in a chain of objects, etc.).

--- **undefined** and **null** have no properties, not even standard methods such as toString().

--- checking for **undefined** or **null**

|  |
| --- |
| **if**(x === **undefined** || x === **null**) { ... } |

you can also exploit the fact the both **undefined** and **null** are considered false:

|  |
| --- |
| **if**(!x) { ... } |

--- **false**, 0, **NaN**, and '' are also considered false

**+++ Categorizing Values Using typeof and instanceof**

There are two operators for categorizing values: **typeof** is mainly used for primitive values, while **instanceof** is used for objects.

**typeof** looks like this:

|  |
| --- |
| **typeof** value |

It returns a string describing the ‘type’ of value. Here are some examples:

|  |
| --- |
| **typeof** **true** // boolean **typeof** 'abc' // string **typeof** {} // object (empty object literal) **typeof** [] // object (empty array literal) |

The following table lists all results of typeof:

Operand Result

undefined 'undefined'

null 'object'

Boolean value 'boolean'

Number value 'number'

String value 'string'

Function 'function'

All other normal values 'object'

(Engine-created value)

JavaScript engines are allowed to create values for which typeof returns arbitrary strings (different from all results listed in this table).

**typeof** **null** returning ‘object’ is a bug that cannot be fixed, because it would break existing code. It does not mean that null is an object.

instance of looks like this:

|  |
| --- |
| vale **instanceof** Constr |

It returns true if value is an object that has been created by the constructor Constr. Here are some examples:

|  |
| --- |
| **var** b = **new** Bar(); // object created by constructor Bar b **instanceof** Bar // true |

|  |
| --- |
| {} **instanceof** Object // true [] **instanceof** Array // true [] **instanceof** Object // true (Array is a sub-constructor of Object) |

|  |
| --- |
| **undefined** **instanceof** Object // false **null** **instanceof** Object // false |

\*\*\* Booleans

The primitive boolean type comprises the values **true** and **false**. The following operators produce booleans:

-- Binary logical operators: && (AND), || (OR)

-- Prefix logical operator: ! (NOT)

-- Comparison operators:

-- Equality operators: ===, !==, ==, !=

-- Ordering operators (for strings and numbers): >, >=, <, <=

+++ Truthy and Falsy

The following values are interpreted as false:

|  |
| --- |
| -- **undefined**, **null** -- Boolean: **false** -- Number: -0, **NaN** -- String: '' |

All other values (including all objects!) are considered true.

Boolean(), called as a function, converts its parameter to a boolean. You can use it to test how a value is interpreted.

|  |
| --- |
| Boolean(**undefined**) // false Boolean(0) // false Boolean(3) // true Boolean({}) // true (empty object) Boolean([]) // true (empty array) |

+++ Binary Logical Operators

Binary logical operators in JavaScript are short-circuiting.

Example, in the following expressions, the function foo() is never called:

|  |
| --- |
| **false** && foo() **true** || foo() |

AND (&&):

If the first operand is falsy, return it. Otherwise,return the second operand:

|  |
| --- |
| **NaN** && 'abc'; // NaN 123 && 'abc' // abc |

OR (||):

If the first operand is truthy, return it. Otherwise, return the second operand:

|  |
| --- |
| 'abc' || 123; // abc '' || 123 // 123 |

+++ Equality Operators

JS has two kinds of equality:

\* Normal, or ‘lenient,’ (in)equality: == and !=

\* Strict (in)equality: === and !==

Therefore, always using strict equality is recommended.

\*\*\* Numbers

All numbers in JS are floating-point:

1 === 1.0; // true

Special numbers include the following:

NaN (‘not a number’)

An error value:

Number(‘xyz’) // NaN (‘xyz’ cannot be converted to a number)

Infinity

Also mostly an error value:

3 / 0; // Infinity

Math.pow(2, 2014) // Infinity (number too large)

Infinity is larger than any other number (except NaN). Similarly, -Infinity is smaller than any other number (except NaN). That makes these numbers useful as default values (e.g., when you are looking for a minimum or a maximum).

\*\*\* Operators

JS has the following arithmetic operators

-- Addition: number1 + number2

-- Subtraction: number1 - number2

-- Multiplication: number1 \* number2

-- Division: number1 / number2

-- Remainder: number1 % number2

-- Incrementon: ++variable, variable++

-- Decrement: --variable, variable--

-- Negate: -value

-- Convert to number: +value

\*\*\* Strings

\*\*\* Statements

\*\*\* Functions

\*\*\* Exception Handling

\*\*\* Strict Mode

\*\*\* Variable Scoping and Closures

\*\*\* Objects and Constructors

\*\*\* Arrays 28

\*\*\* Regular Expressions

\*\*\* Math

\*\*\* Other Functionality of the Standard Library

# Chapter 12: Strings

# Chapter 13: Statements

# Chapter 14: Exception Handling

# Chapter 15: Functions

|  |
| --- |
| **function** **foo**(parameter1, parameter2) {  ... } foo(3, 7); // arguments are used to invoke a function. they are also called actual parameters and actual arguments. in the following example, 3 and 7 are arguments. |

**Defining Functions:**

-- function expression

-- function declaration

-- constructor Function()

functions are objects, instance of Function:

|  |
| --- |
| **function** **id**(x) {  **return** x; } console.log(id **instanceof** Function); // true |

**Function Expressions**

|  |
| --- |
| **var** add = **function**(x, y) {  **return** x + y; } console.log(add(3, 4));// 7 |

**Hoisting:**

Hoisting means 'moving to the beginning of a scope.' Function declarations are hoisted completely, variable declarations only partially.

|  |
| --- |
| console.log(foo(5, 6)); // 11 **function** **foo**(para1, para2) {  **return** para1 + para2; } |

but only the declarations, not assignments

|  |
| --- |
| **var** foo; foo(5, 6); foo = **function**(para1, para2) {  console.log(para1 + para2);  }; // TypeError: foo is not a function |