Javascript

Giuseppe Della Penna

Università degli Studi di L'Aquila
giuseppe.dellapenna@univaq.it
http://people.disim.univaq.it/dellapenna

> *This document is based on the slides of the Web Engineering course, translated into English and reorganized for a better reading experience. It is not a complete textbook or technical manual, and should be used in conjunction with all other teaching materials in the course. Please report any errors or omissions to the author.*

1. Introduction to Javascript

Javascript is a **programming language** commonly used to provide dynamics to web pages.

Technically, it is an interpreted, prototype-based language that supports various programming styles: *object-oriented, imperative, and declarative*.

Javascript is defined based on the **ECMAScript** language specification (latest edition: ECMA-262)

Javascript is not Java : the two languages have very different syntax, semantics, and usage.

Javascript has all the features and constructs common to the most popular **imperative programming languages** (variables, loops, conditional statements, assignment statements, basic mathematical operations, functions and procedures, etc.).

Where can I use it?

Javascript is a very popular language among programmers, due to its versatility and simplicity, and because of this, it has extended far beyond client-side web programming. In fact, Javascript is now used to program a multitude of widely used applications.

- **Client-side web development** : As we know, JavaScript is used in conjunction with HTML and CSS to create the parts of a web page that users see and interact with in their browsers.
- **Server-side web development** : Thanks to the advent of **Node.js**, a commonly used Javascript framework for back-end development, Javascript can also be used for server-side programming.
- **Game Development** : This category, although it falls properly into that of client-side development, is very important and should be highlighted. Previously, a lot of online games were developed with different and/or proprietary technologies such as Flash. Today, Javascript is used to program all the online 2D and 3D games.
- **Mobile application development** : Mobile applications were initially developed natively, i.e. interfacing directly with the operating system (Android, iOS) of the device and using the programming language preferred by it. Currently, frameworks such as **React Native** and **Ionic**, allow you to create cross-platform mobile applications using the same languages as the web: HTML, CSS and of course Javascript.
- **Desktop application development** : following the example of mobile applications, numerous frameworks are emerging, such as **Electron**, which allow you to develop desktop applications using Javascript (and often HTML and CSS to define the interface) instead of the more common languages such as Java or C++.
- Javascript can also be used to add dynamics to **PDF** documents!

1.1. Execution Environment

Javascript is designed to run inside a **host object**, or *environment*, which can extend it by providing additional functionality specific to that environment.

For example, on a web page, the environment, which is the browser, provides JavaScript with functions to **read and modify the HTML structure of the page** and its style sheets (**DOM**), as well as

to **interact with certain parts of the browser** itself, **intercept all actions performed by the user** on the page, and make calls to other servers (**AJAX**)

However, the browser builds a "**sandbox**" around the web page, so that the scripts it contains cannot interfere with other open pages or the rest of the machine.

Browsers also provide Javascript with access to a number of extended functionalities,the so-called **Web APIs** , which extend considerably the "power" of Javascript.

Some examples of Web APIs, whose name is rather self-explanatory: Battery API, Bluetooth API, Canvas API, Clipboard API, Console API, Credential Management API, Fetch API, File System Access API, Fullscreen API, Geolocation API, History API, Payment Request API, Picture-in-Picture API, Screen Capture API, Sensor API, Storage Access API, Web Audio API, Web Authentication API, Web Crypto API, Web Speech API, Web Storage API, WebGL, WebRTC.

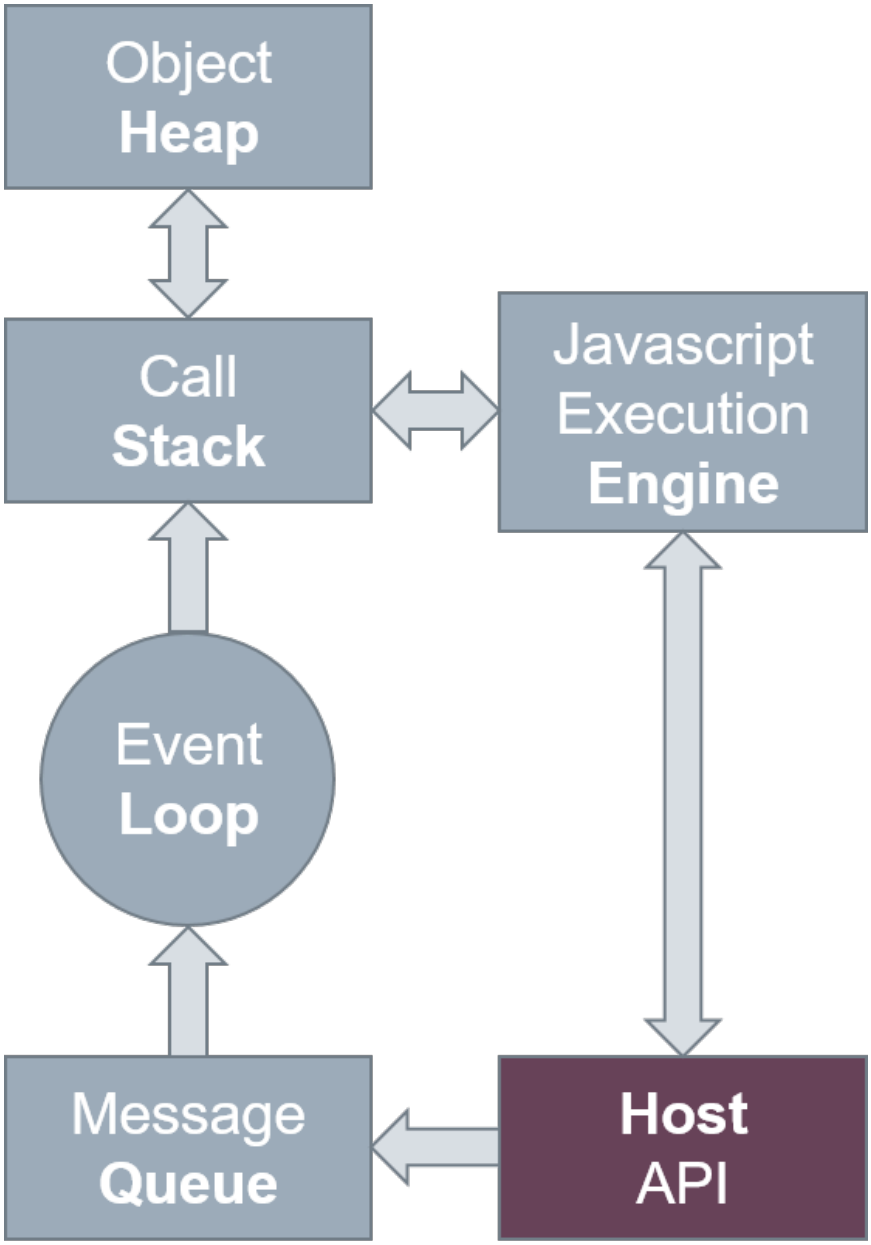
1.2. Execution model

Javascript is a **single-threaded** language, i.e. it does not allow any kind of parallel execution.

The Javascript execution engine extracts the **stack** frames, which correspond to **function calls**, one at a time and executes them. Executing a function can **add new frames** to the top of the stack through **nested calls** to other functions. Objects created during execution are placed on the **heap**.

Javascript code, in its execution, typically interacts with the host's **APIs** (e.g., the browser's *Web APIs*). These APIs can contain **asynchronous** elements (such as the *setTimeout* function or the *DOM events*) that can queue **messages** in the message **queue**. A message is, essentially, a function to be called (**callback**) accompanied by some information about the event that requires its execution.

When the stack is empty, the **event loop** extracts from the message **queue** the next callback function to invoke and places it on the stack.



1.3. Javascript Evolution

ECMAScript 5 (2009) updates

- Strict mode
- Getters and setters for properties (declared in constructors)
- New methods for reading/writing object properties
- New String methods: `trim()`
- New array methods: `isArray()`, `forEach()`, `map()`, `filter()`, `reduce()`, `reduceRight()`, `every()`, `some()`, `indexOf()`, `lastIndexOf()`,...
- `JSON.parse()` and `JSON.stringify()` methods
- New Date method: `now()`

ECMAScript 6 (2015) updates

- Variables and functions with block scope

- Arrow functions (lambda)
- Extended syntax for function parameters
- Spread operator
- String interpolation
- Shorthand properties, Calculated property names
- Destructuring assignment
- Modules
- Classes
- Iterators
- Generators
- Set and Map Structures
- Promises

Recent updates

Here is a summary of some of the most important changes to the standard in recent years.

ECMAScript 2016 (7):

- Exponent operator (**)

ECMAScript 2017 (8):

- Async functions

ECMAScript 2018 (9):

- Finally method in Promises
- Asynchronous Iterators
- Spread in objects destructuring assignment

ECMAScript 2019 (10)

ECMAScript 2020 (11)

ECMAScript 2021 (12)

- Various enhancements not related to the topics covered in these slides. For up-to-date information, see, for example:
  - https://github.com/daumann/ECMAScript-new-features-list
  - https://v8.dev/features
  - https://github.com/tc39/ecma262

Compatibility

All the features introduced with ECMAScript 5, 6 and beyond, as well as all the new features related to the HTML DOM that are gradually adopted by the most modern browsers **are obviously not compatible with older ones**.

When creating a script, it is always necessary to ask yourself **which browsers are the target** (and therefore also the audience of users) and check if the (advanced) features you intend to use are compatible with them. To check the compatibility of a certain function/API, you can search on **MDN** (*https://developer.mozilla.org/en-US/docs/Web*) or sites like *https://caniuse.com/*.

For features with little support in older browsers, it is useful, in order to expand the audience of supported browsers, to **include in your scripts the corresponding polyfills** , i.e. libraries that integrate unsupported features into browsers through appropriate workarounds. Try using the libraries generated by *https://polyfill.io*, which allows you to automatically download polyfills only for the features you are interested in and only if necessary for the browser that requires them.
Alternatively, you can use a transpiler like Babel (*https://babeljs.io*) to transform your scripts in order to make them backwards compatible with specific older browsers. Babel manipulates the syntax and integrates the necessary polyfills.

2. Types, Variables and Operators

2.1. Data Types

JavaScript supports four different data types:

Numbers

- There is no distinction between integers and reals.
- Numerical constants are all the expressions that represent a valid number, with or without decimal point.
- The `parseInt` and `parseFloat` functions can be used to convert strings to numbers.

Booleans

- The boolean constants are *true* and *false*.

Strings

- Strings are special JavaScript objects. They can be created implicitly, through a string constant, or explicitly through the constructor of the String object.
- Values enclosed in quotes (single or double) are string constants.

Objects

- Objects are a very common data type in JavaScript.
- The variables of type object contain *references* to objects. Different variables can then refer to the

same object.

Null

- The null type has only one value, *null*.

2.2. Variables

JavaScript variables are identified by alphanumeric sequences whose first character must be alphabetic.

Variables do not have a type, which is automatically deduced from the value assigned to them, and can change from time to time.

Variables can be declared using `var name`

The initial value of a variable is always the special value `undefined`. It is also possible to initialize the variable in the declaration: `var name = value`.

A value is assigned to an undeclared variable, Javascript creates it automatically. This practice however is **not recommended** because automatically created variables **are always global**.

Trying to read the value of a variable never declared or assigned, returns the *undefined* value.

Examples

```
var o = new Object(); //o is an (empty) Object variable

var s = "pluto"; //s is a String variable with value "pluto"

var n = 3; //n is a Number variable with value 3

m = n; //m is a global variable of type Number with a value of 3

t = "paperino" //t is a global String variable with value "paperino"

u = v //u has undefined value (since v is in turn undefined)

var b = (3>2) //b is a Boolean variable with true value
```

strict mode

Strict mode is activated using the syntax `"use strict";`

Note the backwards compatible syntax: older browsers will see it as a meaningless string

You cannot:

- Use undeclared variables or objects
- Duplicate parameter names in functions
- Assign values to *read-only* or *get-only* properties

In addition:

- The string "eval" cannot be used as a variable
- `eval()` can't create variables in its call scope

block scope

Before ES6, variables declared with the `var` keyword were not global, but *function-scoped*, i.e. accessible throughout the function to which they belong.

In ES6, as in most languages, a variable declared with the new `let` keyword is *block-scoped*, i.e. **visible only in the block** ({...}) in which its declaration is located, and **cannot be redeclared** within the same block.

```
for (let i  = 0; i < a.length; i++) {   let x  = a[i]; … }
//i and x are only available within the for block
```

A more complex example:

```
let callbacks = []; for (let i = 0; i <= 2; i++) {
 callbacks[i]=function(){return i * 2;};
}
//callbacks[0]()=0, callbacks[1]()=2, callbacks[2]()= 4
```

To achieve the same effect of the script above in ES<6, we need to create a closure to fix the value of i relative to each iteration and associate it with the generated callback:

```
var callbacks = [];
for (var i = 0; i <= 2; i++) {
 (function(i) {callbacks[i]=function() {return i * 2;};})(i);
}
```

2.3. Constants

Constants can be declared using the `const` keyword.

```
const PI = 3.141593;
```

Constants are *block-scoped* as variables declared with the `let` keyword.

Note: a *constant object* cannot be reassigned, but its contents can vary since the constant, in such a case, is only the object *reference* assigned to the identifier.

2.4. Operators

`+` **(sum)**
Can be applied to numbers, and to strings, where it represents the concatenation operator. If, in a sum, at least one operand is a string, the others are converted to strings, too.

`-` **(difference)**, `/` **(quotient)**, `*` **(product)**, `%` **(modulus)**

`=` **(assignment)**, `+=`, `-=` **(assignment with sum / difference)**
Assignments with sum / difference have the same semantics as their counterparts in C or Java. The assignment with sum can also be applied to strings, exactly as the sum.

`++` **(increment)**, `--` **(decrement)**

`>>` **(shift right)**, `<<` **(shift left)**, `&` **(and)**, `|` **(or)**, `^` **(xor)**, `~` **(not)**
Perform bitwise operations.

`&&` **(and)**, `||` **(or)**, `!` **(not)**
Used to combine boolean expressions.

`in` **(membership)**
Can be applied to objects or arrays (right operand) to check if they contain the given property or index (left operand).

`>` **(greater than)**, `<` **(less than)**, `>=` **(greater than or equal)**, `<=` **(less than or equal)**, `==` **(equal)**, `!=` **(not equal)**
These operators also work with strings, using the lexicographic ordering.

`typeof(...)` **(type name)**
Returns a string containing the name of the type of its argument

`void(...)` **(void statement)**
Runs the code passed as an argument, but does not return its return value

`eval(...)` **(script evaluation)**
Runs the script passed as a string and returns its value

Examples

```
var s = "tre " + 2; //s is the string "tre 2"

s += " uno"; //s is the string "tre 2 uno"

s > "ciao"; //the expression evaluates true, since the value of s lexicographically follows "c

typeof(s); //returns "string"

var o = {pippo: 1};
"pippo" in o; // the expression evaluates true, since pippo is a property of o

void(0); //null statement (useful in HTML to prevent default actions)
void(f(x)); //executes f(x) and ignored its return value

eval("f(x)"); //executes the script, calling f(x) and returning its return value

eval("3+1"); //returns 4
eval("var s = 1"); //globally declares a variable s and assigns the value 1 to it.
```

3. Flow Constructs

3.1. Conditional execution - if

JavaScript has an `if` statement with the same syntax as Java:

```
if (expression) {
 body
} else {
 body-else
}
```

Non boolean guard expressions are converted to a boolean value as follows:

- If the expression has a numeric value other than zero it is true.
- If the expression has a non-empty string value it is true.
- If the expression has an object value it is true.
- In all the other cases (number zero, empty string, undefined or null) the expression is false.

To execute some instructions only if a specified variable or property is defined and not empty, simply write `if (variable) {...}`

3.2. Conditional execution - switch

JavaScript has a `switch` construct with the same syntax as Java:

```
switch (expression) {
 case v1: statements
 case v2: statements
 default: statements
}
```

The expression is evaluated and compared with the value of each `case`. The statements are then executed *from the first* `case` with the same value as the expression. If no case is selected, the `default` statements are executed, if present.

To stop the execution after a set of statements, it is possible to insert a `break` keyword.

Examples

```
var s = "value";
var b = 0;

//if construct with "mixed" condition
if (s && b > 0) { s = "ok" } else { b = 1; }

//switch constrict on a string
switch (s) {
 case "ok": …
 break; //this case ands here
 case "error": … //this case continues with the default
 default: …
}
```

3.3. Loops

JavaScript has the common loop constructs `for`, `while` and `do ... while`, with the same syntax as Java:

```
for (initialization; condition; update) {body}
```

```
while (condition) {body}
```

```
do {body} while (condition)
```

The `for` loop executes the *initialization*, then if the *condition* is true, it runs the *body* and the *update* instructions. If the *condition* is still true the loop continues.

The `while` loop *body* is executed if and until the *condition* is true.

The `do...while` loop *body* is executed at least once, because the *condition* is tested at the end of its execution.

In the body of the loop you can use the keywords `break` and `continue` respectively to stop the loop or to jump directly to the next cycle.

The for...in and for...of loops

A special form of a `for` loop can be used to loop through all the properties of an object :

```
for (property in object) {body}
```

In each iteration, the string *property* will contain the name of the next property of *object*. You can then access the property by writing `object[property]` (array syntax). Since methods are properties with a particular type, also these will be returned by the loop.

In ES6 there is also another syntax of the `for` construct that allows one to iterate through the values of *iterable* objects, including arrays:

```
for (value of array) {body}
```

In each iteration, the *value* will contain the value of the next array element
We will see later what are iterable objects and how to create them

Examples

```
var o = new Object();

//iterates among object poroperties
for (p in o) {
 o[p]; //gets the value of the property currently pointed in the loop (and should use it…)
}

var i = 0;

//standard for loop
for (i=0; i<10; ++i) { j=j+1; }

//while loop
while(i>0) { i=i+1; }

//do loop
do { i=i+1; } while(i>0);
```

4. Functions

4.1. Declaration

In Javascript, it is possible to create new functions with one of the following syntaxes:

- Function declaration: `function name(parameters) {body}`
- Anonymous functions: `function(parameters) {body}`
- Function objects: `new Function("parameters", "body")`

The different syntaxes have specific characteristics and limitations:

- A function declared *with a name* can be called at any point in the code by its name.
- An anonymous function or a function created with the **Function** constructor should be assigned to a variable (or a property of an object) to be used.

The function **name** can be any valid name for a variable.

The **parameters** of the function, if any, are declared through a list of names (variables), separated by commas. The parentheses after the function name should always be included, even if the parameter list is empty.

The **body** of the function is a sequence of valid JavaScript instructions. Each statement is separated from the next by a semicolon. In the body, parameter values can be manipulated through the variables with the same name.

Arrow functions (lambda)

ES6, like Java 8, allows you to define anonymous functions (lambdas) with a simplified syntax `(parameters) => expression`:

```
(a, b) => a + b //is a valid Function-typed expression
((a, b) => a + b)(1,2)===3 //can be invoked
let f = ((a, b) => a + b) //and assigned
```

In the case of a single parameter, parentheses can also be omitted:

```
a => a + 1 //the increment function
```

For more complex function bodies, you can use the syntax `(parameters) => { body }`:

```
a => {return a+1;}
(x,y) => {a[x]=y;}
```

In the body of the arrow functions, the value of `this` is propagated from the outside:

```
this.nums.forEach(
 (v) => {if (v % 5 === 0) this.fives.push(v); }
);
```

Whereas in ES<6 we should have written

```
var self = this;
this.nums.forEach(
 function (v) { if (v % 5 === 0) self.fives.push(v); }
);
```

Examples

```
//function without parameters, explicit declaration
function f() {
 var i;
}

//function with two parameters, explicit declaration
function g(a,b) {
 var c = a + b;
}

//anonymous function assigned to a variable
var h1 = function(a) {return a+1;}

//function object assigned to a variable
var h2 = new Function("a","return a+1;");
```

block scope

Functions are only visible within the block that defines them (no special syntax is needed):

```
{//block
 function foo () { return 1; }
 foo() === 1;
 { //nested block
  function foo () { return 2; }
  //function redefined in the nested block
  foo() === 2;
 }
 //in the outer block the original version is not overwritten
 foo() === 1;
}
```

4.2. Reference

JavaScript functions are actually variables with a value of type **Function**.

To refer to a function, simply use its name, or an equivalent expression that has a value with type **Function**.

Once the reference to a function is obtained, you can:

- Call the function and pass it some parameters.
- Pass the function as an argument to another function.
- Assign the function to one or more variables.
- Access to all elements of the function, to modify or redefine it, using the properties of the **Function** object.
- Check if a function is defined as you would do with any variable, i.e., writing `if(function_name)`.

4.3. Call

To call a function, append the parameter list, between brackets, to an expression that refers to the function itself:

```
function_name(arguments)
```

```
expression_with_Function_value(arguments)
```

Arguments are a list of valid expressions, separated by commas.

It is possible to omit one or more parameters at the end of the list. In this case, these parameters will have *undefined* value in the function body.

If the function has no parameters, you must still write the two parentheses after its name to call it.

Examples

```
function f() { var i; }
var h1 = function(a) {return a+1;}
var h2 = new Function("a","return a+1;");

f(); //returns undefined

var r = h1(3); //r=4

var r2 = h2(4); //r=5
```

4.4. Passing Parameters

Parameters passing in JavaScript functions takes place in a different manner depending on the type of the parameter itself:

- The types boolean, string, number and null are passed *by value*. The function receives a copy of the value used as argument. Local changes to this value in the function do not affect the value of the argument used in the function call.
- The type Object is passed *by reference*. The manipulation of such parameters inside the function are reflects on the objects used as an argument to the function call.

Passing Parameters - extended syntax

In ES6 you can specify **default values** for parameters (such as in PHP or C):

```
function f(x, y=7, z=42) {...}
```

It is also possible to define **variadic functions** , i.e. functions with variable arity, as in other languages (C, Java, etc.), specifying a last parameter (indicated with the syntax `...variable`) in which all the *other* arguments passed to the function (*rest parameters*) will be inserted, in the form of an array:

```
function f( ...args ) {return args.length;}
```

```
function f (x,...a) {return x*a.reduce((t,v,i,r) => a+v, 0);}
```

4.5. Returning

Functions return the control to their caller at the end of their instructions block.

It is possible to *return a value* to the caller using the syntax

```
return expression
```

The *expression* can be of any type. It is evaluated and the resulting value is returned.

If the function does not execute any return statement, JavaScript implicitly issues a `return undefined` at the end of its code.

4.6. Closures

A **closure** is technically an expression (typically a function) *associated with a context that assigns its free variables*.

All the Javascript code is executed in a context, including the global one.

In particular, each execution of a function has an associated context.

A *closure* is created by a function, when it returns a new function, created dynamically (i.e., with one of the three constructs seen previously).

Behavior of closures

A *closure*, i.e., a function created within another function and then returned, *maintains the execution context of the function that created it*.

This means that the context of each call to the "generator" function is not destroyed when the function ends, as generally happens, but it is stored in memory.

The *closure* may refer to (read/write) the parameters and variables declared in the context of the function that generated it.

Since each function call has its own distinct environment, the values "seen" by the *closure* will not be affected by subsequent calls to the generator function.

Closures - Examples

A common use for the closure is to provide parameters to a function that will be executed later, for example for the functions passed as an argument to `setTimeout` (which will be described later).

If we pass a function as an argument, or assign it to a variable, we can not provide it with parameters, but instead we can use a *wrapper closure* that calls it with the desired parameters.

See the following examples ...

```
function f(x) {
 return x+global_variable;
 //NOTE: the return value does NOT depend only on the parameter values
}
//we want to assign a property p of o with the FUNCTION returning f(3)
o.p = f(3);
o.p();
//ERROR: o.p is not a function, but the return value obtained
//by calling f(3) when the previous instruction was executed

o.p = f;
o.p();
//ERROR: o.p is a reference to f, thus is must be called
//with a parameter (e.g., we should write o.p(3))

function closureGenF(y) {
 return function() {return f(y);}
}
o.p = closureGenF(3);
o.p() //CORRECT: f(3) will be called!
```

```
//we want to assign an handler for the onclick event to an HTML DOM element
htmlelement.onclick = f;
//NOTE: we cannot pass parameters here, as in the previous example

//Often it happens that the same handler can be used for different elements,
//only with small adjustments.
//Such variants are easy to construct at runtime (and it is often necessary).
//For example, we want to associate to a set of elements an handler which
//sets their color to red when clicked

function clickHandler(oToHighlight) {
 return function(e) {
  oToHighlight.style.backgroundColor="red";
 }
}

element1.onclick = clickHandler(linkedelement1);
element2.onclick = clickHandler(linkedelement2);
```

5. Objects

Javascript **is not object oriented language similar to more well-known ones (as Java)**, and his concept of object is much more similar to an associative array.

Javascript objects contain methods, which can however be considered values, too, as they are merely objects of class **Function**.

In Javascript it is not possible to define **classes**, but only special special **constructors** that create objects with certain members. The name of the constructor function is referred as the class name of the generated objects.

There is no real inheritance in JavaScript objects, and you can not declare hierarchies. However, JavaScript contains a default base class called **Object**.

Objects are created using the **new** operator applied to their **constructor function**: `o = new Object();`

An alternative method to create an object is to use the construct `{"property": value, ...}`, which creates an object with the given properties assigned to the corresponding values.

5.1. Properties

Javascript object properties can be assigned to values of any type.

To reference a property, two different syntaxes can be used:

- "object oriented" syntax: `object.property`
- "array" syntax: `object["property"]`

The special construct `for...in` can be used to iterate between the object properties.

It is possible to check if an object contains a property using the boolean expression `property in object`.

If you try to read the value of an undefined object property, *undefined* will be returned, as for any unassigned variable. It is possible to **dynamically add properties** to any object by simply assigning them a value.
However, it is not possible to add properties to variables that are not of type object (predefined objects or created with `new`)

Properties - Examples

```
var o = new Object();

var v = o.pippo;
//v is undefined

o.pluto = 3;
//now o has a "pluto" property, with Number type and value 3

v = o.pluto;
//now v is a Number variable with value 3

v.paperino = "ciao";
//ERROR: properties can be added only to variables having Object type

var o2 = {"pippo": "ciao", "pluto": 3};
//implicit (short) object creation

v = o2["pluto"];
//same as a v =o2.pluto

var nome = "pippo";
//now nome is a String variable with value "pippo"

v=o2[nome];
//access to a property with a dynamically calculated name,
//assigned to the nome variable
```

5.2. Methods

Methods are simply a Javascript object properties of type **Function**. *Function* is a predefined JavaScript object, and can be used directly, for example to create anonymous functions.

To **access** a method you can use the same syntax used for the properties.

To **call** a method simply append the parameter list, in brackets, to the expression which accesses the method.

To add a method to an object, simply create a property with the name of the method and assign to it:

- A function already defined
- An anonymous function: `function(parameters) {body}`

Methods can be added at any time to an object, just as the properties.

Methods, *to refer to the properties of the object they are defined in*, must use the `this` keyword: `this. property`.
If you omit `this`, JavaScript will search for a variable with that name within the method or between the global variables!

Methods - Examples

```
var o = new Object();

o.metodo1 = function(x) {return x;}
//adds the function to the object as metodo1

o["metodo2"] = f;
//adds the function f (if exists) to the object as metodo2

o.metodo1
//this expression returns the Function object representing meotodo1

o.metodo1(3);
o["metodo1"](3);
//two equivalent calls to metodo1

var o2 = {"pippo": "ciao", "pluto": 3, "metodo3": function(x) {return x;}}
//method definition in the short object creation syntax

var o3 = new Object();
o3.metodo3 = o.metodo1
//metodo3 of object o3 is a copy of metodo1 in object o
```

Simplified property definition

It is possible to use a compact syntax for **defining properties from variables**: the object's property can automatically take on the same name as the variable used to assign it a value:

```
let x=0, y=0;
let o={x,y}; //same as {"x":x, "y":y}
```

Similarly, when defining a property associated with a method, you can **write the method directly**, without the *function* keyword, which will be assigned to a property of the same name:

```
let o={s(x,y){return x+y}};
//same as {s: function(x,y){return x+y}} or {s: (x,y)=>{return x+y}}
```

You can also define properties with a **calculated name** using the `[name-expression]` syntax:

```
let o={p:"ciao", ["p_"+f()]: 42 };
```

5.3. Constructor Functions

A constructor function is a special kind of function where:

- You use the `this` keyword to *define* the properties of a new object.
- There is no return value

The constructor functions should be used as an argument for the `new` operator, just like the standard JavaScript object names:
`new function(parameters)`

The constructor functions should never be called directly.

When calling the constructor with `new`, JavaScript creates an empty object derived from **Object** and applies it to the function.

In the constructor, `this` points to the new object. In this way, the constructor can populate the new object, adding properties and methods through `this`. Note again that the methods of an object, to refer to the properties of the same object, must use the `this` keyword.

Examples

```
function myObject(a) {
 this.v = a+1;
 this.w = 0;
 this.m = function(x) {return this.v+x;}
}

/*
 The object will have two properties, v and w, the former initialized through
 the constructor parameter a, and a method m, which returns the value of
 property v added to its parameter
*/

var o = new myObject(2);
o.m(3); //returns 6;

o.getW = function() {return this.w;}
//dynamic addition of members to an instance object (NOT to the constructor)

o.getV = function() {return v;}
//ERROR! v points to the GLOBAL variable v!
```

5.4. Prototypes

When a Javascript object is created using `new`, it is implicitly assigned to a *prototype*.

Prototypes contain **methods and properties common to all the objects created with the same constructor**, and thus are roughly equivalent to the concept of *class* in other languages.

When an object is extended with new properties or methods, these are **only added to that specific object**.

However, if you **extend the prototype of an object**, the extensions will be **available in all the objects created by its own constructor, even earlier**.
In a sense, Javascript allows to dynamically extend classes.

When trying to access an object's property, it is first searched for in the object itself, then in its prototype. As the prototype itself is an object, this procedure is repeated until it reaches the prototype of *Object*, the base object. Thanks to this effect, and since the prototype of an object can be dynamically reassigned, it is possible to achieve in Javascript something similar to the inheritance hierarchies.

The prototype of the objects created by a constructor can be examined and modified by accessing its `prototype` property. If we have an instance object, we can access its prototype by first going to its constructor function through the `constructor` property, or by using the (less supported) `__proto__` property.

The `instanceof` operator allows one to check whether a constructor function is present anywhere in the prototype chain of an object.

Examples

```
function myObject(a) {
 this.v = a+1;
 this.w = 0;
 this.m = function(x) {return this.v+x;}
}
var o1 = new myObject(1);
var o2 = new myObject(2);

o1.m(1); //returns 3
o2.m(1); //returns 4

o1.z = function() {return this.v;}
//member dynamically added to object o1

o1.z(); //returns 2
o2.z(); //ERROR, z is only in o1!
```

```
//all the following expressions point to the
//prototype of the objects created by the myObject constructor
o1.constructor.prototype == o1.__proto__; //true
o1.__proto__ == myObject.prototype; //true:

//member dynamically added to all the objects created
//by the myObject constructor
myObject.prototype.x = function() {return this.v+2;}

o1.x(); //returns 4
o2.x(); //returns 5

o1 instanceof myObject //returns true
o1 instanceof Array // returns false
[1,2,3] instanceof Array // returns true
```

5.5. Getters and Setters

In Javascript, **getters** and **setters** allow you to create pseudo variables in an object. A pseudo variable is not directly associated with a real object variable, but it works as if it were.

Actually, getters and setters are methods that are automatically called when you try to read or write the variable represented by their name.

In this way, the read and write operations are mediated by the code:

- It is possible to calculate a value in the getter "on the fly"
- It is possible to check or correct the value to be assigned in the setter (maybe throwing an exception if the value is unacceptable)
- You can make a value get-only or set-only

Getters and setters can be defined in individual instances:

```
var o = {
 val:1,
 set pippo(v) {if (v<10) this.val=v; else throw "wrong value"},
 get pippo() {return this.val}
 //r/w property called pippo, only accepts values less than ten
}
```

To add them to constructors, you need to use the new `defineProperty` method on their prototypes instead, as we will see later

Object.defineProperty method

`Object.defineProperty(object, propertyName, descriptor)` inserts or overwrites a property in an

object.

- *descriptor* is an object that can contain the following properties
  - *configurable*: *true* if the descriptor can be changed
  - *enumerable*: *true* if this property is included in the object's property enumeration
  - *value*: the value of the property
  - *writable*: *true* if the value of the property can be altered (false, the default, makes it read-only)

```
var o={a:2}
Object.defineProperty(o, "val1", { value: 3, writable:true});
//property "val" R/W ... same as if we wrote o.val=3
Object.defineProperty(o, "val2", { value: 4, writable:false});
// property "val" R/O
```

As an **alternative** to *value* and *writable*, you can use the *get* and *set* properties to create a pseudo variable:

```
Object.defineProperty(o, "pippo", { get : function() { return this.val1 }});
//pseudo property "pippo", read only view on the val1 property defined above
```

To define getters and setters **in a constructor**, you can extend its prototype:

```
function obj() {this.val=4};
Object.defineProperty(obj.prototype, "pippo", {
 get : function() { return this.val },
 set : function(v) { this.val=v }
});
//getters and setters defined in the prototype of a constructor Function
```

5.6. Public, private and "privileged" members

In Javascript there is no explicit notion of public and private member, common to many object oriented languages.

However, you can **simulate this behavior** by using the techniques just exposed.

We will see how to define properties (and methods) so that they are visible outside the object or can only be used by its internal methods.

Public members

Public properties and methods can be created as we have seen so far, i.e., assigning them to the `this` object within the constructor.

However, Javascript's *coding standards* suggest to proceed as follows:

- Properties are **created within the constructor** function
- Methods are **added to the prototype of the constructor**

The final effect is the same but, as we will see, it has some impact on private properties.

Public members - Examples

```
function myObject(a) {
 this.v = a+1; //public property
}
myObject.prototype.m = function(x) {return this.v+x;} //public method

var o = new myObject(1);
```

Private members

Private properties and methods can be created by exploiting the constructor's *closure* effect

In practice, any property or method declared in the constructor function **as a local variable** (that is, with the keyword `var` or `let`, and not with `this`) will be a private member of the objects it generates.

Private methods **can access public members of the object** but, to overcome an ECMAScript ambiguity, this cannot be done using the common syntax `this.p`: a workaround is required.
We declare a private property (which we shall call `THIS`) and we assign it to the value of `this` within the constructor. Private methods can then access public members with the `THIS.p` syntax.

Private properties and methods **are not accessible from the outside of the object** but, unlike common object-oriented languages, **they are not accessible also by public methods** created as seen before. Therefore private properties and methods can only be manipulated by other private methods or by the constructor in which they are declared!

Examples

```
function myObject(a) {
 this.v = a+1; //public property
 var p = 7; //private property
 var THIS = this; //workaround to make this available to private methods
 var pm = function() {p=p-1;} //private method
 var pm2 = function() {return THIS.m()+1;} //private method calling a public method
}
myObject.prototype.m = function(x) {return this.v+x;} //public method

var o = new myObject(1);
o.p; //undefined (p is private)
o.pm() //ERROR (pm is private)

myObject.prototype.m2 = function() {pm(); return p;}
//public method using private members

o.m2(); //ERROR, since p e pm cannot be accessed by public methods
```

Privileged members

Being unable to access private members from public methods makes private members useful only for internal functions, such as initialization.

However, **it is possible to create special public methods that also have access to the object private members** and which, for this reason, are often called *privileged*.

In practice, many prefer to always declare privileged methods rather than public methods to achieve a better symmetry with the standard behavior of object oriented languages.

Creating privileged methods is very simple: just use the "base" technique for creating methods explained before, i.e., **define them directly in the constructor** (and not add them to the prototype) and the closure will do the rest.

Examples

```
function myObject(a) {
 this.v = a+1; //public property
 var p = 7; //private property
 var THIS = this; //workaround to make this available to private methods
 var pm = function() {p=p-1;} //private method
 var pm2 = function() {return THIS.m()+1;} //private method calling a public method
 this.m3 = function() {pm(); return p;} //privileged method: is public and uses private member
}
myObject.prototype.m = function(x) {return this.v+x;} //public method

var o = new myObject(1);
o.p; //undefined (p is private)
o.pm() //ERROR (pm is private)

myObject.prototype.m2 = function() {pm(); return p;}
//public method using private members

o.m2(); //ERROR, since p e pm cannot be accessed by public methods

o.m3(); //returns 6
```

5.7. Classes

ES6 introduces a new class concept, similar to that of object-oriented languages such as Java or C++.

The syntax for defining a class is the common one, which involves the use of the `class` keyword.

An ES6 class can only contain methods, declared without the `function` keyword. Other class properties are initialized in the constructor by assigning them, as in the prototype system.
The constructor is declared as a method with the reserved name `constructor`.

```
class Shape {
 constructor(id, x, y) {
  this._id = id;
  this.move(x, y);
 }
 move(x, y) {
  this._x = x;
  this._y = y;
 }
}

s = new Shape("test",1,2)
```

A class can be declared by its name or assigned as a *class expression* to an identifier.

```
let Shape =  class { constructor(id) { this._id = id;} }
```

Special methods

You can use the `static` keyword to declare methods that can be invoked directly on the class, and not on its instances:

```
class Shape {
 constructor(id, x, y) {
  this._id = id;
  this.move(x, y);
 }
 move(x, y) {
  this._x = x;
  this._y = y;
 }
 static defaultShape() {
  return new Shape("default", 100, 100);
 }
}

s = Shape.defaultShape();
```

It is possible to declare *getters* and *setters* with the ES5 syntax:

```
class Shape {
 constructor(id, x, y) {
  this._id = id;
  this.move(x, y);
 }
 move(x, y) {
  this._x = x;
  this._y = y;
 }
 static defaultShape() {
  return new Shape("default", 100, 100);
 }
 set x(x) { this._x = x; }
 get x()   { return this._x; }
}
```

Inheritance

You can generate **class hierarchies** using the `extends` keyword. In derived classes, the `super` keyword provides access to the methods and constructor of the base class.

```
class Rectangle extends Shape {
 constructor (id, x, y, width, height) {
  super(id, x, y);
  this._width=width;
  this._height=height;
 }
 toString() {
  return "Rectangle > " + super.toString();
 }
}
```

There are no visibility modifiers (*private*, *protected*,...): if necessary, you can use the techniques already illustrated for prototypes to achieve similar effects.

Examples

```
// Classes are not a substitute for the prototype-based system of previous
// versions. Actually, classes are only syntactic sugar, and are internally
// mapped on the old object management system.

class Shape {
 constructor(id, x, y) {this._id = id; this.move(x,y);}
 move(x, y) { this._x = x; this._y = y; }
 static defaultShape () { return new Shape("default", 100, 100); }
 get x() { return this._x; }
}
//SAME AS
let Shape = function(id,x,y) {
 this._id = id;
 this.move = function(x, y){ this._x = x; this._y = y; }
 this.move(x, y);
}
Shape.defaultShape = function(){return new Shape("default", 100, 100);}
Object.defineProperty(Shape.prototype,"x",{get: function(){return this._x}}
```

6. Spread and Destructuring Assignment

6.1. Expression expansion (spread)

It is possible to use the *spread* operator (denoted by `...`, not to be confused with the syntax of *rest parameters*) to **explode an array (or *any iterable object*) in the sequence of its values**

- during a **function call**

```
var p = [1,2];
function f(x,y) {return x+y}
r=f( ...p); //r=f(1,2).
//r=f(p) does not work!
```

- and in an **array assignment**:

```
var a = [ "a", "b",  ...p ]; //a=[ "a", "b", 1, 2 ]
var str="ciao"; var chars = [...str]; //chars=["c","i","a","o"]
//the last example works since String is iterable!
```

6.2. Destructuring assignment

Destructuring assignment allows you to **extract array values or object properties and assign them to separate variables**.

With arrays , the syntax is to use an expression in square brackets on the *left* side of an assignment:

```
var list = [ 1, 2, 3 ];
var  [ a, b, c ] = list; //a=1, b=2,c=3
```

It is possible **not to assign part of the elements** of the array:

```
var  [ a, b ] = list; //a=1, b=2
var  [ a, , b ] = list; //a=1, b=3, note the commas
var [ , , a ] = list; //a=3
```

Or assign the rest of the array using a **rest parameter**:

```
var  [ a, ...b ]  = list; //a=1, b=[2,3]
```

With objects, the syntax is to use an expression in curly brackets on the left side of an assignment:

```
var o={p1:1, p2:"a", p3(x){return x+1}};
var  {p1, p2, p3}=o; //p1=1,p2="a",p3=x=>x+1
```

You can **extract only part of the object** by specifying only the properties you need:

```
var  {p3}=o; //p3=x=>x+1
```

You can **give extracted variables a name** other than the object's property by using the syntax

`property: variable`

```
var  {p1:a,p3:b}=o; //a=1, b=x=>x+1
```

It is also possible to **extract the same property several times** with different names:

```
var {p1:a,p1:b}=o; //a=1, b=1
```

Destructuring can be invoked **recursively**:

```
var o2={p1:1, p2:"a", p3:{s1:2, s2:3}};
var {p3:a, p3:{s2:b}}=o2; //a={s1:2, s2:3}, b=3
```

It is possible to provide **defaults in the destructuring** , so that missing elements are assigned to the default and not *undefined*:

```
var [a, b, c, d]=list //d=undefined
var [a, b, c, d=10] = list //d=10
var {p4}=o; //p4=undefined
var {p4=1} =o; //p4=1
```

Destructing can also be used intelligently when **passing parameters to functions** (which is seen as assigning arguments to parameter variables):

```
function f ([a,b]) {return a+b;}
f(["Ciao",10]); //returns "Ciao10"
//in this case f("Ciao",10) does not work

function g ({p1:a,p2:b}) {return a+b;}
g({p1:"Ciao",p2:10}); //returns "Ciao10"
```

7. Iterators and Generator Functions

7.1. Iterators

Definition

ES6 objects can customize how they are "enumerated" using the for construct, which has a new syntax specifically for this purpose.

An iterable object implements a method whose name is the value of the expression `Symbol.iterator`.

This method must return an object that implements the interface of an iterator, which must contain a `next` method which, in turn, returns an object with the two properties `value` (the next value of the iterator) and `done` (*true* if the values are ended)

```
let numbers1_10 = {
 [Symbol.iterator]: function() {
  let n = 1;
  return {
   next: function() {
    if (n<=10) return { done: false, value: n++ };
     else return { done: true, value: undefined };
   }
  };
 }
}
```

Of course, it is possible to dynamically construct an object and its iterator via a function:

```
function numbers(a,b) {
 return {
  [Symbol.iterator]() {
   let n = a;
   return {
    next() {
     if (n<=b) return { done: false, value: n++ };
     else return { done: true, value: undefined };
    }
   };
  }
 }
}
```

Use

An iterable object can be enumerated using its iterator, for example with a for loop:

```
it =  numbers1_10[Symbol.iterator]();
for(let n=it.next(); !n.done; n = it.next()) {
 console.log(n.value)
}
```

As we have seen, we can use the `for...of` statement specific to iterable objects:

```
for(n of numbers1_10) console.log(n)
```

The **spread** operator and **destructing assignment** also work with any iterable object:

```
let [a,b,,d] = numbers1_10 //a=1, b=2, d=4
[0,...numbers1_10] //an array from 0 to 10
```

Many Javascript objects are iterable, such as arrays and strings, but also Map and Set that we will see later.

7.2. Generators

Since defining an iterator is quite complex, ES6 provides another useful construct for creating iterable objects with calculated and, theoretically, unlimited values: generator functions.

Generator functions are special functions that freeze their execution returning a value, and when they are invoked, they resume execution from where they left off.

A generator is defined as a normal function by placing an asterisk after the function keyword (`function *`).

Inside the generator, the generated values are returned with the `yield` keyword. To complete the iteration, you just need to exit the method.

```
function *   g_numbers1_10() {
 for(let i=1; i<11; ++i) {
  yield i;
 }
}
```

Calling a generator function returns an instance of the corresponding iterable object:

```
numbers1_10 = g_numbers1_10()
for(n of numbers1_10) console.log(n)
```

In fact, we can also write `numbers1_10[Symbol.iterator]()` to fetch the iterator corresponding to the generator, and manipulate it as seen above.

Generators can also be defined anonymously (function expressions) and as object methods:

```
var g_numbers1_10 = function *() {for(let i=1; i<11; ++i) yield i;}
var o = {n:0, g_numbers1_10:function *() {for(let i=1; i<11; ++i) yield i}}
```

8. Exceptions

Newer versions of JavaScript also introduced a **Java-style exception handling system**.

An exception reports an *unexpected situation*, often a *mistake*, within the normal execution.

An exception may be raised by libraries or by JavaScript code written by the user, through the `throw` keyword.

To handle exceptions, you can use the `try ... catch ... finally` construct.

Handlers

Once raised, an exception goes back on the JavaScript *call stack* until it is handled. This means that an exception thrown in a function, if not handled within it, will be propagated to its caller functions, until it gets to the JavaScript *runtime*.

To handle exceptions generated by a block of code, you must insert the block inside the `try ... catch` construct. Any exception raised in the code between `try` and `catch` will be passed to the error handling code declared after the `catch`.

If you want to ensure that a piece code is *always* executed after the `try ... catch` block to be protected, regardless of possible exceptions, you can add a `finally` clause to the block.

Example

```
try {
 … code…
} catch(e) {
 //exceptions generated by Javascript are objects whose message property
 //contains the error message
 alert("Exception raised: "+ e.message);
}

try { …code… } catch (exception) {…exception handling…}
finally {
 ...code executed in any case before the execution point goes after the try block...
}
try {
 …code…
 throw {"name": "pippo", "value": 1};
 //we can raise an exception with any object
} catch (ex) {
 //the object ex in the catch block is the one thrown with the throw clause
}
```

9. Predefined Objects

9.1. String

String objects in JavaScript are used to contain strings of characters. They can be created implicitly, using a string constant, or explicitly through the constructor:

`s = new String(value)`

The main methods and properties of the **String** class are as follows:

- `length`
  returns the length of the string.
- `charAt(position)`
  returns the character (string of length one) at the given *position* (zero based).
- `charCodeAt(position)`
  as charAt, but returns the ASCII code of the character.
- `indexOf(s, offset)`
  returns the position of the first occurrence (from *offset*, if specified) of *s* in the string. Returns -1 if s is not a substring of the given string (starting from offset).
- `lastIndexOf(s, offset)`
  as indexOf, but returns the last occurrence.
- `substr(os[, l])`
  returns the substring of length *l* (default, the maximum possible) that starts *os* characters from the beginning of the string
- `substring(os, oe)`
  returns the substring that begins at *os* characters and ends at *oe* characters from the beginning of the string
- `toLowerCase()`
  returns the string converted to lowercase
- `toUpperCase()`
  returns the string converted to uppercase

String interpolation

ES6 allows you to construct strings by interpolating them, i.e. inserting variable values into them without having to use concatenation (as happens, for example, in PHP).

To make interpolation possible, the string must be delimited by the special character ` **(which is not the apostrophe ')**

```
var persona = {nome:"Pippo"};
var saluto="Hello";
`${saluto} ${persona.nome}!`; //Hello Pippo!
```

It is also possible to insert expressions in the interpolations:

```
var prodotto = { quantita: 7, nome: "gelato", prezzounitario: 3 };
`You ordered ${prodotto.quantita} ${prodotto.nome}, for a total price of ${prodotto.quantita *
```

9.2. RegExp

JavaScript uses regular expressions written in Perl syntax.

To write a constant regular expression it is sufficient to use the syntax `/*expression*/`.

Regular expressions variables can be created using the **RegExp** constructor.

It is possible to use regular expressions in various methods of the **String** class:

- `match(r)`
  returns the array of substrings that match the regular expression *r*.
- `replace(r, s)`
  replaces all the substrings that match *r* with the string *s*.
- `search(r)`
  returns the position of first substring matching *r*, or -1 if there is no match.
- `split(r [, m])`
  splits the string into a series of segments separated by the separators specified by *r* and returns them as an array. If you indicate a maximum length *m*, then the last element of the array will contain the remaining part of the string.

By default, JavaScript interrupts the process of regular expression matching just after the first match. To find all the possible matches, use the `/g` modifier

To make the expression and *case insensitive*, use the `/i` modifier

9.3. Array

Arrays are predefined JavaScript objects and can contain values of any type.

To **create** an array you can use one of the following syntaxes:

- (multiple parameter constructor) `v = new Array(e1, e2, e3, ...)`
- (symbolic constructor) `v = [e1, e2, e3, ...]`

To **access** an element of an array, use the common syntax `array_variable[index]`

It is possible to check if an index is present in an array using the boolean expression `index in array_variable`.

The main methods and properties of the **Array** class are as follows:

- `length`
  returns the size of the array
- `concat(e1, e2, e3,...)`
  adds the items at the end of the array.
- `join(separator)` converts the array into a string, concatenating the **String** version of each element with the given *separator* (default ",").
- `reverse()`
  reverses the order of the array. Warning: in addition to returning the sorted array, this function modifies the source array. If you don't want to modify it, you can create a copy.
- `slice(os [, l])`
  returns the sub-array of length *l* (default, the maximum possible) which starts at index *os*.
- `sort([sortfun])` sorts the array. The optional *sortfun* can be used to specify a non-standard sort order. Warning: in addition to returning the sorted array, this function modifies the source array.

Examples

```
var a1 = new Array(10,20,30);
//declaration using the new construct

var a2 = ["a","b","c"];
//implicit declaration

for (i in a1) { a1[i]; }
//iterates in the array (considering also all its other properties and methods!)

for (i=0; i<a1.length; ++i) { a1[i]; }
//iterates among the array elements

if (4 in a1) { a1[4] = a1[4]+1; }
//use an array element only if it is defined

/*
Note: to create an associative array, you can simply dynamically create new properties (array
*/
```

Arrays – Further methods

- `forEach(function (value, index, array) {...} )`
  executes a code on each array element. The given function receives the *value* of the current array element, its *index* and the *array* itself
- `map(function (value, index, array) {return …} )`
  creates a new array with the same number of elements mapped using the function
- `filter(function (value, index, array) {return …} )`
  creates a new array with only the elements for which the function returns true

- `reduce(function (aggval, value, index, array) {return …}, init)`
  returns a value obtained by aggregating the elements of the array through the passed function, which receives, in addition to the current element, also the current value of the aggregate (*aggval*, which initially is *init*) and updates it, returning it
- `every(function (value, index, array) {return …} )`
  returns true if the function is true on all the array elements
- `some(function (value, index, array) {return …} )`
  returns true if the function is true on at least one of the array elements
- `indexOf(v,start)` / `lastIndexOf(v,start)`
  returns the first/last index of element *v* in the array, optionally starting from the *start* index
- `find(function (value, index, array) {return …} )`
  returns the first element for which the function is true
- `findIndex(function (value, index, array) {return …} )`
  returns the index of the first element for which the function is true

9.4. Date

The **Date** object allows one to manipulate values of type date and time. It has several constructors:

- `Date()` initializes the object to the current date/time.
- `Date(y, m, d, hh, mm, ss)` initializes the object with the date/time *d/m/y hh:mm:ss*.
- `Date(string)` tries to recognize the *string* as a date and initializes the object accordingly.

Date objects can be compared with each other with the normal comparison operators.

The Date object methods allow you to read and write all the members:

For example, `getYear`, `getMonth`, `setYear`, `setMonth`, `getDay` (returns the *day of the week*), `getDate` (returns the *day of the month*), `setDate` (sets the day of the month: if the passed value is greater than the maximum allowed, *the function automatically increases the month/year*)

Examples

```
//builds a greeting based on the current hour of the day and assigns it to the saluto variable
var giorni = ["lun","mar","mer","gio","ven","sab"];
var mesi = ["gen","feb","mar","apr","mag","giu","lug","ago","set","ott","nov","dic"];
var oggi = new Date();

var data = giorni[oggi.getDay()] + " " + oggi.getDate() + " " + mesi[oggi.getMonth()] + " " +o

var saluto;
if (oggi.getHours() > 12)
 saluto = "Good evening, today is "+data;
 else saluto = "Good morning, today is "+data;


//gets a date 70 days in the future
futuro = new Date();
futuro.setDate(futuro.getDate()+70);
```

9.5. Set

ES6 provides two new **container classes** (alongside Array) present in the libraries of all object-oriented languages: **Map** and **Set**.

Set represents an unordered set of (unique) elements. Once you have created a Set (`new Set()`) you can:

- **Add** elements with the `add()` method
- **Remove** elements with the `delete()` method
- **Empty** the set using the `clear()` method
- Read the **cardinality** from the `size` property
- **Check** for an element with the `has()` method
- **Enumerate** elements (Set is iterable)

```
let s = new Set();
s.add(4).add(1).add(2).add(3);
s.delete(2);
s.has(2) === false;
s.size === 3
for (let v of s) console.log(s);
```

9.6. Map

Map represents an association between keys and values. **Keys and values can be of any type (even objects)**. Once you have created a Map (`new Map()`) you can:

- **Create** new associations or update existing ones with the `set()` method

- **Read** the value associated with a key with the `get()` method
- **Remove** an association with the `delete()` method
- **Empty** the map with the `clear()` method
- Read the **cardinality** from the `size` property
- **Check** for a key with the `has()` method
- **Enumerate** keys using the iterable returned by the `keys()` method, values using the iterable returned by the *values()* method, and key-value pairs using the map itself as an iterable

```
var m = new Map()
m.set("a",1).set(4,"four").set(o,34); //o is an object
m.has("a")===true;
m.delete("a");
m.get(4)==="four";
for(v of m) console.log(v);
for(v of m.keys()) console.log(v);
for(v of m.values()) console.log(v);
```

9.7. Promise

ES6 promises **replace callback functions** as a way to return values (or errors) asynchronously. A promise is the result of a computation that will end in the future.

In other words, a promise allows you to use an asynchronous function as if it were synchronous because this function, while working *asynchronously*, will immediately return (*synchronous* behavior) a value, which is not the one actually calculated, but a *promise* (a *proxy*), through which the final value can be obtained once the processing is finished.

The `Promise(executor)` constructor is invoked by passing as a parameter an *executor* function of the type `(resolve, reject) => {...}` where *resolve* and *reject* are also functions.

The *executor* code **represents the (possibly asynchronous) processing associated to the Promise**.

If the processing finishes successfully, it calls the *resolve* function passing it the calculated value, otherwise it can call the *reject* function passing an error value.

Caution: **the *executor* code does not run asynchronously unless you use asynchronous functionalities within it** . In other words, Promises **don't create parallel threads** .

```
p = new Promise((resolve,reject) =\> {
 //something long and asynchronous
 resolve(10); //result 10
 //or reject(message);
})
```

Handlers

While it's possible to create Promises as we have just seen, it's more common to use Promises returned by some asynchronous Javascript APIs.

The Promise code **starts as soon as the Promise is created**, even if it doesn't yet have the *resolve* and *reject* handlers associated with it. The latter **will be called as soon as they are set**, based on the final state of the Promise.

Note: In many cases, creating a promise whose code calls reject will throw an exception with the value passed to the reject itself if there is no specific handler associated with this event.

The handlers, i.e. functions that correspond to *resolve* and *reject*, can be specified using the `then()`, `catch()` and `finally()` methods of the Promise object.

- `then(f1,f2)` executes *f1* if the Promise terminates successfully, or *f2* if it ends with an error. The *resolve* argument from the Promise will be passed as a parameter to the *f1* function. The *reject* argument from the Promise will be passed as a parameter to the *f2* function.
- `then(f)` is equivalent to `then(f,null)`
- `catch(f)` is equivalent to `then(null,f)`
- `finally(f)` is equivalent to `then(f,f)`, but in this case the function *f* will **not** receive any value (neither the one passed to *resolve* nor the one passed to *reject*)

```
p.then(m=>{console.log("Resolved: "+m)}); //resolve only

p.then(
 m=>{console.log("Resolved: "+m)},
 m=>{console.log("Rejected: "+m)}); //resolve and reject

p.finally(
 ()=>{console.log("done")}); //finally only
```

Examples

```
function doMyWorkAsync(x) {
 return new Promise((resolve, reject) => {
 //we use setTimeout to simulate an asynchronous execution
  setTimeout(function(){
   //At the end of the (asynchronous) code of the promise,
     //if everything went well, we call the resolve function-parameter
     //and pass the result of the computation to it
   resolve(x+1);
     //In case of failure, we do the same with the reject function
  }, 250);
 });
}

//As soon as you create a promise, its executor code starts
v_proxy = doMyWorkAsync(1);
//however, calls to resolve and reject are placed on hold
//until you specify the corresponding handlers

//With the then method of the promise, we specify what to do
//with the return  value, when available
v_proxy.then((v) => {
 console.log("Returned value: " + v);
});
```

Chaining

It is often useful to *concatenate* asynchronous operations, i.e. to start an asynchronous operation when the previous one ends, passing the result of the latter as an argument.

With promises, this is possible because the `then` and `catch` methods return a new promise, which represents the completion of the execution of the respective handlers.

In the body of a handler used in a chain, the `return` will correspond to a call to *resolve* with its value, while an exception (`throw`) will be transformed into a call to *reject*.

In this way, you can call a series of concatenated `then`, specifying in each one what to do when the previous operation ends.

Examples

```
function doMyWorkAsync(x) {
 return new Promise((resolve, reject) => {
  setTimeout(function(){
   if (x>=0) resolve(x+1);
   else reject("x is negative")
  }, 250);
 });
}

doMyWorkAsync(1)
.then(
 function(y) {
   //the handler computation may, in turn, take some time
   return y+1;
 }
).then( //this handler is executed when the previous one returns
 function(z) {
  console.log(z);
 }
).catch(
 function(e) {
  //this handler is executed when an exception is thrown in the
  //chain or reject is called
  console.log("Error: "+e);
 }
).finally(
 function() {
  console.log("Complete");
 }
)
```

9.8. Async Functions

Asynchronous functions, declared through the `async function` construct, are a method of *handling the creation of Promises and/or the code dependent on the fulfillment of one or more Promises in a more linear and simple way* .

An async function, when called, returns a **Promise for its result** (regardless of its actual return value).

A **return in the async function corresponds to a call to the resolve function** of the promise with that value as an argument.

Any exception raised in the async function corresponds to a call to the reject function of the promise with the exception as an argument.

The body of the function is the body of the Promise.

The following two blocks of code, therefore, are equivalent:

```
function pf1(x) {
 return new Promise((resolve,reject) => {
  if (x>=0) resolve(x);
  else reject("x negative");
 });
}
pf1(1).then((v)=>{console.log(v)},(e)=>{console.log("Error: "+e)});

async function pf2(x) {
 if (x>=0) return x; //resolve
 else throw "x negative"; //reject
}
pf2(1).then((v)=>{console.log(v)},(e)=>{console.log("Error: "+e)});
```

await and implicit handlers

Within an *async function* it is also possible to **synchronize** in a simplified way with other Promises, without using the `then` and `catch` constructs, but working synchronously with the `await` construct.

With the syntax `v = await p`, where *p* is a Promise, the function code is **suspended** waiting for *p* to finish.

The value passed to the *resolve* of *p* will be **assigned to the variable *v***, whereas the value passed to the *reject* of *p* will become **an exception** raised by the `await`

Therefore, await "moves" the asynchrony of the nested Promise (*p* in the example) onto that of the async function that contains it, making the Promise on which it is executed synchronous.

For this reason, **await can only be used as an async function**: while the async function remains suspended on the await, it is not actually a blocking for the script, as it automatically returns a Promise itself.

Example

```
function pf1(y) {
 return new Promise((res,rej) => {
  if (y>=0) res(y+2);
  else rej("y negative");
 });
}

async function pf2(x,y) {
 if (x>=0) {
  let z = await pf1(y);
  return x+z;
 }
 else throw "x negative";
}

pf2(1).then(
 (v)=>{console.log(v)},
 (e)=>{console.log("Error: "+e)});
```

The pf1(y) function returns a Promise.

The asynchronous function **pf2(x,y) waits for the Promise of z=pf1(y) to be resolved** and then returns the sum x+z.

The Promise returned by pf2(x,y)

- **calls reject** (second `then` parameter) if pf2 throws an exception, either explicitly (`throw`) or for a *reject* call of pf1 on which pf2 performs an `await`.
- **calls resolve** (first `then` parameter) if pf2, after pf1 calls its *resolve*, returns its value normally with a `return`.

10. Javascript in Browsers

10.1. Scripts in HTML pages

Embedding

To embed a script in an HTML page, use the <script> tag with type attribute set to the value "text/javascript".
For compatibility with older browsers, the code inside the script tag is sometimes placed between HTML comment tags: <!-- and -->.

It is also possible to load an external script, leaving the <script> tag empty and specifying the URI of the script in the src attribute.
For compatibility with some browsers, never write the script tag in the abbreviated form, but always put inside it some text, e.g. an empty comment (/* */).

Since the script may contain reserved XML characters, it should always be enclosed in a **CDATA section**. For compatibility with some browsers, it is necessary to comment (with //) the lines containing the opening and closing tags of the CDATA section.

It is possible to insert or import several different scripts to inside the same document.

There are also some attributes allow to embed code in the HTML page:

- The events attributes such as *onclick* may contain pieces of code (not statements), to be executed when the event occurs.
- The href attribute of the tag <a> can refer to a Javascript function with the syntax: "javascript:funcname(arguments)". In this case, the click of the link will perform the function call.

Execution

The <script> tag can appear both in the <head>, where it is normally placed, or at any point of the <body>.

All functions and variables declared in the script become available (i.e., can be referred in the code) as soon as the parser analyzes the page fragment that declares them.

If a script contains immediate code, i.e., written outside functions, it is executed when the parser analyzes the page fragment that contains it. In this way, for example, it is possible to ensure that a script is evaluated only after the HTML element it refers to is loaded.

Scripts can freely use functions and variables declared in other scripts included in the same page.

Examples

```
<script type="text/javascript">
//<![CDATA[
 var s = "pluto";
//]]>
</script>

<script type="text/javascript" src="script.js">
/* */
</script>
```

10.2. Window object

When JavaScript is used in a browser, there are other useful objects that can be accessed, including the browser itself and the displayed page.

The **window** object is the access point for all the other objects exposed by the browser. This is *the default object* for scripting, i.e., all its properties and methods are available at global scope, without

explicitly specifying the window object.

The interface of window contains some very useful features, including

- The `alert(message)` method, which shows the given *message* in a dialog box (with the OK button only).
- The `confirm(message)` function, which shows the given *message* in a dialog box with the OK and Cancel buttons. The function returns true if the user clicks OK, false otherwise.
- The `prompt(message, default)` method, which displays the given *message* in a dialog box, together with an input field with *default* as initial value. If the user clocks OK, the input field contents (even if empty) are returned by the function. Otherwise the function returns null.
- The `setTimeout` and `setInterval` methods allow to create timers (see later)
- The `document` property provides access to the displayed HTML document
- Other properties, such as `statusbar`, have still a very browser-specific semantics

Window object - Examples

```
//executes an action only if the user clicks OK
if (window.confirm("Are you sure you want to leave the page?")) {…}

//asks the user to enter an information and alerts him if an empty value was entered
var citta = window.prompt("Place of birth","L'Aquila");
if (!citta) window.alert("You have not specified the place of birth!");
```

10.3. Timers

Javascript, through the **window** object, allows to execute **timed actions**. For this purpose it is possible to use the following methods.

- `setTimeout(string_or_function, milliseconds, arg1, ..., argn)` Calling this function ensures that, after the specified number of *milliseconds*, JavaScript executes the code given by the first argument, which can be a *string* containing the code to be evaluated or the name of a *function* to call. In the latter case, you can *optionally* specify a number of arguments (*arg1* ... *argN*) to be passed to the function. **The action is performed once**.
- `setInterval(string_or_function, milliseconds, arg1, ... argn)`. Calling this function ensures that every *millisecond*, JavaScript executes the code given by the first argument, which can be a *string* containing the code to be evaluated or the name of a *function* to call. In the latter case, you can *optionally* specify a number of arguments (*arg1* ... *argN*) to be passed to the function. **The action is executed periodically**.

Both functions can be called multiple times, and return a *timer id* (numeric), through which you can cancel the timer using the corresponding functions:

- `clearTimeout(id)` for the timers initiated with `setTimeout()`

- `clearInterval(id)` for the timers initiated with `setInterval()`

Timers - Examples

```
function saluta(nome) {
 alert("Hello "+nome);
}

//Asks the user name and greets him after 5 seconds
var nome = prompt("What is your name?");
if (nome) setTimeout(saluta,5000,nome);


//notices the current time every minute
setInterval("d=new Date(); alert('Now it's '+d.getHours()+':'+d.getMinutes())",60000);
```

10.4. Document object

The **document** object can be retrieved as a property of the **window** object and represents the document displayed by the browser.

Most of the methods and properties provided by the document object are described in the **Document** interface, which will be discussed as part of the *Document Object Model*. However, there are some useful properties which are present only in the document object, for example

- The `location` property contains the URL of the current document.
- The `lastModified` property contains the last update date of the document.
- The `open()` method opens a stream to write text in the document via `write()` and `writeln()`. When the stream is opened, the current contents of the document are deleted.
- The methods `write(text)` and `writeln(text)` append *text* (or *text* followed by a carriage return) to the current document. If you did not call the `open()` before, it is implicitly called before the first write or writeln is issued. **Warning**: *These methods can not be used with XHTML*.
- The `close()` method closes the stream opened by `open()` and forces the display of what has been written to the document with `write()` and `writeln()`. Any subsequent write operation will generate a new implicit `open()`.

The document object often provides also a browser-specific system to access the structure of the displayed document. In modern browsers, with support for the W3C DOM, the use of this system is however strongly discouraged.

Document object - Examples

```
//creates a document containing a simple table
var i,j;
document.open();
document.write("<table border='1'>");
for(i=0;i<10;++i) {
 document.write("<tr>");
 for(j=0;j<10;++j) {
  document.write("<td>"+i+","+j+"</td>");
 }
 document.write("</tr>");
}
document.write("</table>");
document.close();
```

10.5. XMLHttpRequest object

The **XMLHttpRequest** object, originally introduced by Internet Explorer, is now supported by all popular browsers.

Its purpose is to *allow Javascript code to send HTTP requests to the server* (just as a browser would do) and use the resulting data.

This object is the basis of **AJAX** techniques, which allow the scripts on a web page to exchange data with the server without the need to "change page".

For safety reasons, the XMLHttpRequest object can *only* make connections *with the host that owns the page* where the script is hosted, unless the contacted host returns appropriate *CORS headers*.

XMLHttpRequest: instantiation

The XMLHttpRequest interface is standard, but there are browser-dependent systems to access this object.

If the browser defines a constructor with the same name (`typeof XMLHttpRequest! = "Undefined"`), simply issue a `new`:
`var XHR = new XMLHttpRequest()`

As an example, in Internet Explorer, we had to use the ActiveXObject object (`typeof ActiveXObject != "Undefined"`) with the string identifying the object to create, which can be "MSXML2.XmlHttp.6.0" (preferred) or "MSXML2.XmlHttp.3.0" (old versions of the browser):
`var XHR = new ActiveXObject ("MSXML2.XmlHttp.3.0")`

XMLHttpRequest: use

The *usage pattern of XMLHttpRequest* is two-fold, depending on whether you choose to make the call synchronous or asynchronous:

- **Synchronous mode**: the request is blocking, i.e., the script (and the associated page) are unavailable until the response is received.
- **Asynchronous mode**: the request is sent and the script continues its execution. The script is then notified of the response through an *event*.

XMLHttpRequest: synchronous use

Prepare the request using the `open` method, passing the url and the HTTP verb to call. The third parameter must be *false* to start a synchronous request:

```
xhr.open("GET", "http://pippo", false);
```

Send the request with the `send` method, which is blocking:

```
xhr.send(null);
```

Check whether the request returned an HTTP error using the `status` property, for example

```
if (xhr.status == 200) {...}
```

Access the data returned by the server (if necessary) using the `responseText` property

XMLHttpRequest: synchronous use - Example

```
var req = createRequest();

req.open("GET",requrl,false);

req.send(null);

if (req.status==200) {
 alert(req.responseText);
} else {
 alert("error");
}
```

XMLHttpRequest: asynchronous use

Prepare the request using the `open` method, passing the url and the HTTP verb to call. The third parameter must be *true* to start a synchronous request:

```
xhr.open("GET", "http://pippo", true);
```

Set up one or more of the following *handlers* (others are available, but we won't use them here):

- the handler to call to monitor the progress of the request: `onreadystatechange` property
- the handler to call when the request has been served (regardless of the resulting state): `onload` property
- the handler to call if the request cannot be served (typically due to a network error): `onerror` property

```
xhr.onreadystatechange = function() {…}
xhr.onload = function() {…};
xhr.onerror = function() {…};
```

Send the request with the `send` method (the call immediately returns the control to the script):

```
xhr.send(null);
```

Within the declared handlers

- Once the request has been served (handler `onload`), you can check whether it returned an HTTP error via the `status` property and then access the data returned by the server via the `responseText` property, as already illustrated.
- In the case of the `onreadystatechange` handler, you can check whether the request has been served (`readyState` property equal to 4) and if so proceed as above.
- In case of error (handler `onerror`), the values of the above properties will not be available.

At any time, you can invoke the `abort` method to stop the current HTTP request.

XMLHttpRequest: asynchronous use - Example

```
var req = createRequest();

req.open("GET",requrl,true);

req.onreadystatechange = function () {
 if (req.readyState==4) {
  if (req.status==200) {
   alert(req.responseText);
  } else {
   alert("server status: "+req.status);
  }
 }
};

//or...
req.onload = function () {
 if (req.status==200) {
  alert(req.responseText);
 } else {
  alert("server status: "+req.status);
 }
};

req.onerror = function () {
 alert("Network Error");
};

req.send(null);
```

XMLHttpRequest: asynchronous use - example with promise

```
let p = new Promise((success, failure) => {
 let req = createRequest();
 req.open("GET", requrl, true);
 req.onload = function() {
  if (req.status==200) {
   success(req.responseText);
  } else {
   failure("server status: "+req.status);
  }
 };
 req.onerror = function() {
  failure("Network error");
 };
 req.send();
});

p.then(
 function(t){alert(t);}
 function(e){alert("Problem: "+e);}
);
```

XMLHttpRequest and JSON

When exchanging data with a script via XMLHttpRequest, it often happens to have the server send *complex data structures* to JavaScript, not simply HTML or plain text.

In these cases, it is useful to use the **JSON** notation: in practice, the data structures are encoded using the "short" JavaScript notation for the definition of objects and arrays.

For example, the string that follows defines (and creates in JavaScript) an array containing two records with fields "id" and "name":

```
[{"id": 1, "name": "foo"}, {"id "2," name ":" bar "}]
```

Once Javascript receives this information as text, it is possible to turn it into the corresponding real data structures with a statement like this:

```
data = new Function ("return" + xhr.responseText)();
```

Or, in ES>=5, using the JSON object methods:

```
data = JSON.parse(xhr.responseText)
```

10.6. Fetch API

Fetch APIs are a generic interface for retrieving resources, even over the network, and in this way they also replace and enhance the *XMLHttpRequest* object, making it unnecessary to use libraries such as JQuery to make complex AJAX calls.

The `fetch` method is implemented by the *window* object and requires the path to the resource you want to retrieve as a required argument

optionally, you can also pass an object with advanced options, such as `method`, `mode` (no-cors, cors, same-origin), `cache` (default, no-cache, reload, force-cache, only-if-cached), `headers`, `redirect` (manual, follow, error), `referrerPolicy` (no-referrer, no-referrer-when-downgrade, origin, origin-when-cross-origin, same-origin, strict-origin, strict-origin-when-cross-origin, unsafe-url), and `body`.

fetch returns a *Promise* that becomes the response (*Response* object) to the request.

There are methods to handle the content of the fetch-generated *Response*, such as `text` or `json` These methods return an additional Promise that resolves to the text or json structure contained in the *Response*.

For further information, see https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Example

```
fetch(requrl, {
  method: 'GET',
  mode: 'no-cors' //you can disable CORS if required!
  headers: {
   "Accept": "application/json"
  }
 }
).then(response => {
  if (!response.ok) { //chacks that response.status is 2xx
     throw new Error('Error'); //generic error
  }
  return response.text(); //promise for response text
}).then(responseText => alert(responseText)
).catch(error => {
  //handles network errors and errors generated by handlers
  alert(error);
});
```

11. Modules

11.1. Modules: exporting

Each javascript file can **represent a module**.

You can **explicitly export** values from modules using the `export` keyword.

Export can refer to values **already declared elsewhere**, and export multiple values in a single statement:

```
export {f1, val3, f2}; //f1, f2 and val3 must be defined
```

It is possible to **rename the exported values** during export:

```
export {f1, val3 as v3  , f2};
```

You can export **the values of variables or functions declared in the same statement**:

```
//file: "path/modulo1.js"
export function inc (x) { return x + 1 };
export var zero = 0;
```

It is also possible to define a **single default export** for each module (whose import will be performed differently, as we will see):

```
export {f1 as default, f2, val3}
export default function  f() {...}
```

Finally, it is possible to **export from a module values imported "on the fly" from a secondary module**:

```
//file: "path/modulo2.js"
export from "path/modulo1.js";
export {inc} from  "path/modulo1.js";
export {inc as add1} from  "path/modulo1.js";
```

11.2. Modules: importing

All values exported from one module can be imported into another one by using the expression `import *` and assigning them a *namespace* (or rather an object that will contain them):

```
// file: "main.js"
import * as m1 from "path/modulo1.js";
m1.inc(m1.zero);
```

It is possible to **import only some values** exported from a module, using a syntax similar to that of

destructuring assignment:

```
// file: "main.js"
import {inc} from "path/modulo1.js";
inc(1);
```

Finally, you can **rename a value during import** :

```
import {inc as add1, zero as base} from "path/modulo1.js";
add1(base);
```

If a module has a **default export**, you can import it directly with a local name by typing:

```
import v from "path/modulo1.js";
//in this way v will be the alias of the default value exported by modulo1.js
```

You can **import the default along with explicit elements**:

```
import v, * as m1 from "path/modulo1.js";
import v, { inc as add1 } from "path/modulo1.js";
```

11.3. Modules in HTML pages

Although modules are useful for building complex Javascript programs outside the browser (for example, for node.js), web pages can also benefit from organizing code into modules. However, you have to keep in mind that the modules don't work like classic scripts:

By default, the modules are in **strict** mode. Modules have their own **scope**. So if you define a global variable in a module, it won't be visible to other modules/scripts

The `import` and `export` keywords are only available in modules

It is therefore necessary to let the browser know that you want to use a block of Javascript as a module, and not as a normal script, in order to enable these features, using the special "**module**" type:

```
<script type="module src="main.js"></script>
```

To provide a **fallback** solution for older browsers, you can also insert a compatible, non-modular script with the `nomodule` attribute. Browsers that know how to manipulate modules will ignore it:

```
<script nomodule src="fallback.js"></script>
```

Obviously, only the **top-level modules** must be imported into the web page, i.e. those that will then import the other modules: the download of the imported modules is triggered automatically through the imports

The **immediate code** present in any module (top or imported) will be executed by the browser when the corresponding file is downloaded.

A module **will never be downloaded more than once**, even if it is imported at various points.

When specifying the path of a module within imports, you need to provide an absolute or relative URL that allows the browser to download it. Warning: **many browsers do not support relative paths such as "lib.js"** (the file to be downloaded is in the same directory as the resource that is importing it): you must instead write "./lib.js"

12. References

ECMAScript 2023 Language Specification
https://tc39.es/ecma262

MDN Web Docs
https://developer.mozilla.org

13. Examples

The main samples explained or developed during the classes are outlined below. These examples are all available in the GitHub, at the address https://github.com/orgs/WebEngineering-Univaq, and are a *key component* of the classes itself, since they illustrate the practical use of the concepts presented during lectures and reported in this documentation (where, when possible, references to these examples can also be found).

The list below may not always be up to date: in the repository you may often find useful new examples that have just been developed.

- JS_Example_simpletooltip
  Standard, CSS-based and Javascript-based degrading tooltips
- JS_Example_simplechecker2
  Javascript form validation library
- JS_Example_simpleordering
  Javascript table ordering script
- JS_Example_simplepager

Javascript table pagination script

- JS_Example_simpleswitchededitor
  Javascript-powered smart input controls
