

# Javascript – the language

## 6 – Taking a look at functions

```
function f(x) {return x*x}
```

# Functions

---

```
<script>  
function add(x,y) {return x+y;}  
function multiply(x,y) {return x*y;}  
function operate(op,x,y) {  
    return op(x,y);}  
document.write(operate(add,3,2));  
</script>
```

Output: 5

View also [https://www.w3schools.com/js/js\\_functions.asp](https://www.w3schools.com/js/js_functions.asp)

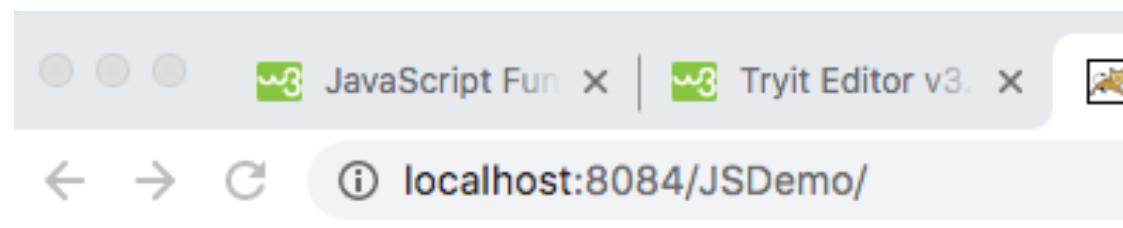


# Functions

```
<!DOCTYPE html>

<html>
<body>
<script>
function toCelsius(fahrenheit) {
    return (5/9) * (fahrenheit-32);
}
document.write(toCelsius);

</script>
</body>
</html>
```



<HTML>

<HEAD>

<SCRIPT>

```
function fact(n) {  
    if (n==1) return n;  
    return n*fact(n-1);  
}
```

</SCRIPT>

</HEAD>

<BODY>

<H2>Table of Factorial Numbers </H2>

<SCRIPT>

```
for (i=1; i<10; i++) {  
    document.write(i+"!="+fact(i));  
    document.write("<BR>");
```

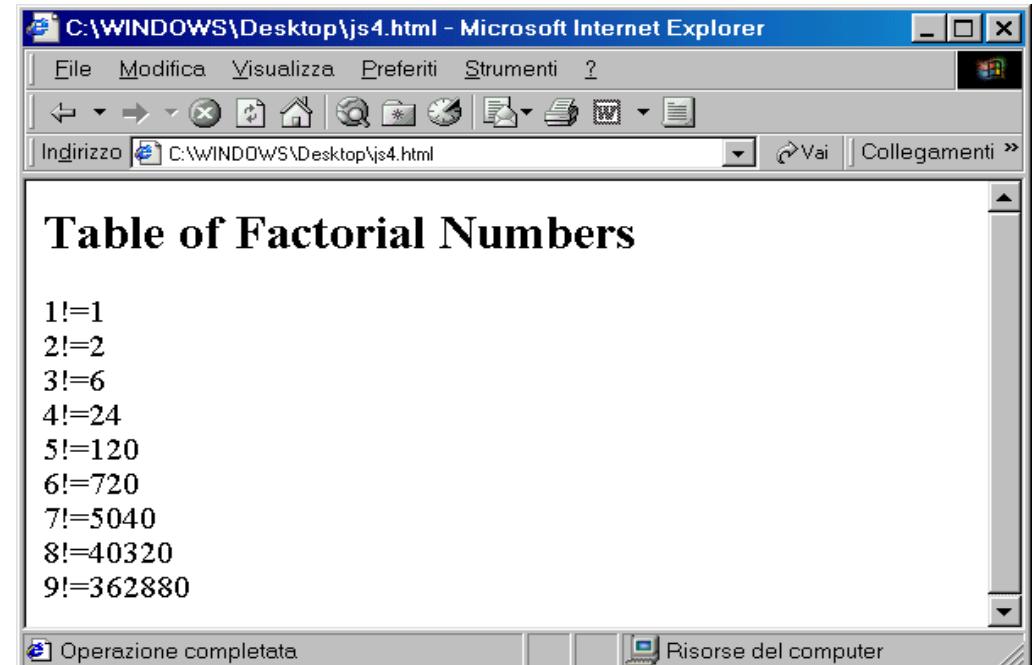
}

</SCRIPT>

</BODY>

</HTML>

# Recursive f.



# Function hoisting

Hoisting is a JavaScript mechanism where **variables** and **function declarations** are moved to the top of their scope before code execution.

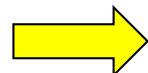
The hoisting mechanism **only moves the declaration**. The assignments are left in place.



# Function statements

The function statement declares a function.

```
hoisted();
```



OUTPUT:  
This function has  
been hoisted

```
function hoisted() {  
  console.log('This function has been hoisted.');//  
};
```

## Function declaration are hoisted



# Function expressions

A JavaScript function can also be defined using an **expression**.

A function expression can be stored in a variable:

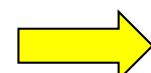
```
var x = function (a, b) {return a * b};
```

After a function expression has been stored in a variable, the variable can be used as a function: **product=x(2,3)** ;

Functions stored in variables do not need function names, as they are always invoked (called) using the variable name.

```
var fundef = function() {  
    document.write('Hello');  
};
```

```
fundef();
```



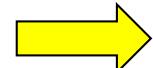
OUTPUT:  
Hello



# Function expressions and hoisting

**Function expressions** load only when the interpreter reaches that line of code.

```
fundef();
```



**OUTPUT:**

```
"TypeError: expression is not  
a function"
```

```
var fundef = function() {  
    console.log('This will not work.');//  
};
```

**Function expressions are not hoisted**



# Arrow functions

- `function multiplyByTwo(num) { return num * 2; }`

## Possible redefinitions:

- `const multiplyByTwo=function (num){ return num * 2; }`
- `const multiplyByTwo= (num) => { return num * 2; }`
- `const multiplyByTwo= num =>{ return num * 2; }`
- `const multiplyByTwo= num => num * 2;`

Usage (in all cases) : `multiplyByTwo(4);`



# Mapping and filtering functions

```
<script>  
twodArray = [1,2,3,4];  
document.write( twodArray );  
document.write( "<BR>" );  
document.write( twodArray.map( num => num * 2 ) );  
</script>
```

OUTPUT:  
1,2,3,4  
2,4,6,8

```
<script>  
twodArray = [1,2,3,4];  
document.write( twodArray );  
document.write( "<BR>" );  
document.write( twodArray.filter( num => num % 2 == 0 ) );  
</script>
```

OUTPUT:  
1,2,3,4  
2,4



# More examples

```
<script>  
multiplyByTwo=function(num) { return num * 2; }  
document.write( multiplyByTwo(6));  
document.write("<BR>");  
myArray=[1,2,3];  
document.write(myArray.map(multiplyByTwo));  
</script>
```

OUTPUT:  
12  
2,4,6,



# Javascript – the language

## 7 - getting deeper on variables

# Undeclared variables

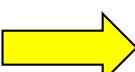
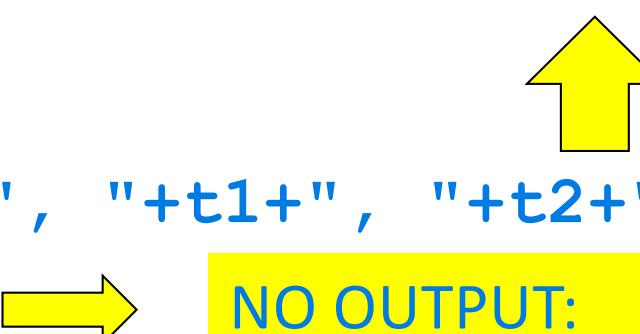
In JavaScript, an undeclared variable is assigned the value "undefined" at execution and is also of type "undefined".

a **ReferenceError** is thrown when trying to access a previously undeclared variable.

```
<script>
    var t0=typeof(x);
    var x=3;
    t1=typeof(x);
    x="pippo";
    var t2=typeof(x);
    document.write(t0+, "+t1+, "+t2+"<br>");
    document.write(z);
</script>;
```

OUTPUT:  
undefined, number, string

NO OUTPUT:  
reference error



# Variable scope 1

Type of declaration	Scope	Note
<code>x=10 ;</code>	always global	
<code>var x=10 ;</code>	function scope	(global if external to any function)
<code>let x=10 ;</code>	block scope	ES 6



# Variable scope 2

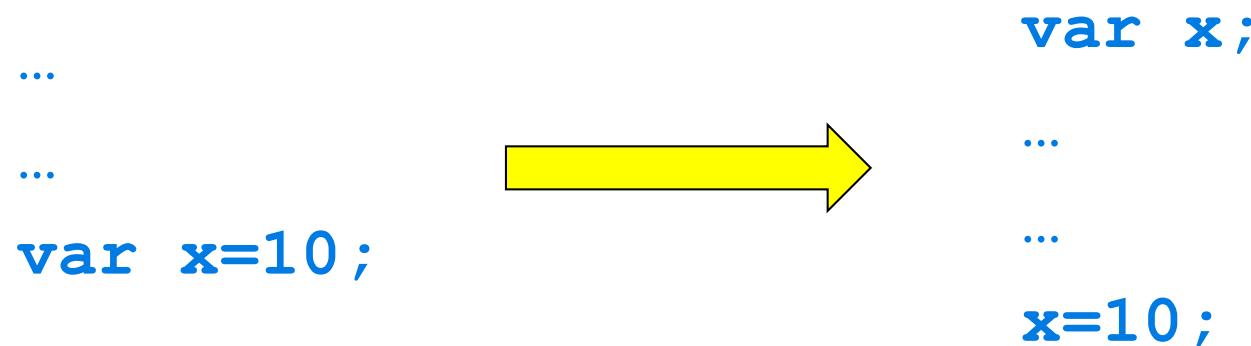
- Variables declared **with var** live in their **function scope** (which, if outside any function, is **global**)
- Variables declared with **let** have **block scope** instead of function scope (ES 6)
- Variables declared **without var or let** are **always global**.



# Variable hoisting

Hoisting is a JavaScript mechanism where **variables and function declarations** are moved to the top of their scope before code execution.

The hoisting mechanism **only moves the declaration**. The assignments are left in place.



Variable declarations are processed before any code is executed.



# Variable scope

```
{ var x = 2; }  
// x CAN be used here
```

```
{ let y = 2; }  
// y can NOT be used here
```



# Variable scope

```
try {  
    p2 = (n,s) => document.write(n+":"+s+"<br>");  
    p1 = (n) => document.write(n+"<br>");  
    // code here can NOT use carName  
    function f() {  
        var carName="volvo"  
        q(carName); // code here CAN use carName  
    }  
    f();  
    p1(carName); // code here can NOT use carName  
} catch (err) {  
    p2("ERROR",err.message);  
}
```

OUTPUT:  
volvo  
ERROR:carName is not defined



# Variable redefinition

```
var x = 10;  
// Here x is 10  
  
{  
    x = 2;  
    // Here x is 2  
}  
  
// Here x is 2
```

```
var x = 10;  
// Here x is 10  
  
{  
    var x = 2;  
    // Here x is 2  
}  
  
// Here x is 2
```

```
var x = 10;  
// Here x is 10  
  
{  
    let x = 2;  
    // Here x is 2  
}  
  
// Here x is 10
```



# Variable scope - example 1

```
try {  
    p2 = (n,s) => document.write(n+":"+s+"<br>");  
    p1 = (n) => document.write(n+"<br>");  
    function f() {  
        a = 20;  
        var b = 100;  
    }  
    f();  
    p1(a);  
    p1("<hr>");  
    p1(b);  
} catch (err) {  
    p2("ERROR",err.message);  
}
```

OUTPUT:

20

-----

ERROR : b is not defined



# Variable scope - example 2

```
try {  
    p2 = (n,s) => document.write(n+":"+s+"<br>");  
    p1 = (n) => document.write(n+"<br>");  
  
    function f() {  
        p1(a);    ➔ NO OUTPUT:  
        a = 20;  
        var b = 100;  
  
    }  
    f();  
} catch (err) {  
    p2("ERROR",err.message);  
}
```



# Variable scope - example 3

```
try {  
    p2 = (n,s) => document.write(n+":"+s+"<br>");  
    p1 = (n) => document.write(n+"<br>");  
  
    function f() {  
        p1(b);    → OUTPUT:  
        a = 20;  
        var b = 100;  
    }  
    f();  
}  
} catch (err) {  
    p2("ERROR",err.message);  
}
```

because of hoisting!



# Variable scope - example 4

```
try {  
    p2 = (n,s) => document.write(n+":"+s+"<br>");  
    p1 = (n) => document.write(n+"<br>");  
  
    p1(b);  
  
    function f(){  
        a = 20;  
        var b = 100;  
    }  
    f();  
} catch (err) {  
    p2("ERROR",err.message);  
}
```

NO OUTPUT:  
ERROR: b is not defined



```
<script>
```

## Variable scope - example 5

```
p2 = (n,s) => document.write(n+": "+s+"<br>");  
x=null; // DEF 1  
  
function f() {  
    var x = "A"; // DEF 2  
  
    p2(2,"x in f "+x);  
    {  
        var x=1; // DEF 3  
  
        p2(3,"x in inner block in f "+x);  
    }  
  
    p2(4,"x in f "+x);  
}  
  
p2(1,x);  
f();  
p2(5,x);  
</script>
```

### OUTPUT:

```
1: null  
2: x in f A  
3: x in inner block in f 1  
4: x in f 1  
5: null
```



# Variable scope - example 5 B

DEF 1	DEF 2	DEF 3	P1	P2	P3	P4	P5	
<pre>x=null let x=null var x=null</pre>	<pre>var x="A"</pre>	var x=1	null	A	1	1	null	
		x=1	null	A	1	1	null	
		let x=1	null	A	1	A	null	
	<pre>x="A"</pre>	var x=1	null	A	1	1	null	WHY?
		x=1	null	A	1	1	1	
		let x=1	null	A	1	A	A	
	<pre>let x="A"</pre>	var x=1			ERROR			(*)
		x=1	null	A	1	1	null	
		let x=1	null	A	1	A	null	

(\*) A let variable cannot be redefined with a larger scope in an inner block.

*Why then I can put let in def 1 without problems?*



```
<script>          Variable scope - example 6
p2 = (n,s) => document.write(n+":"+s+"<br>") ;
function f() {
    x = "A"; // DEF 1
    p2(1,"x in f "+x);
    {   p2(2,"x in inner block in f "+x);
        let x=1; // DEF 2
    }
    p2(3,"x in f "+x);
}
try {
f();
} catch(err) {
    p2("ERROR",err.message);
} </script>
```

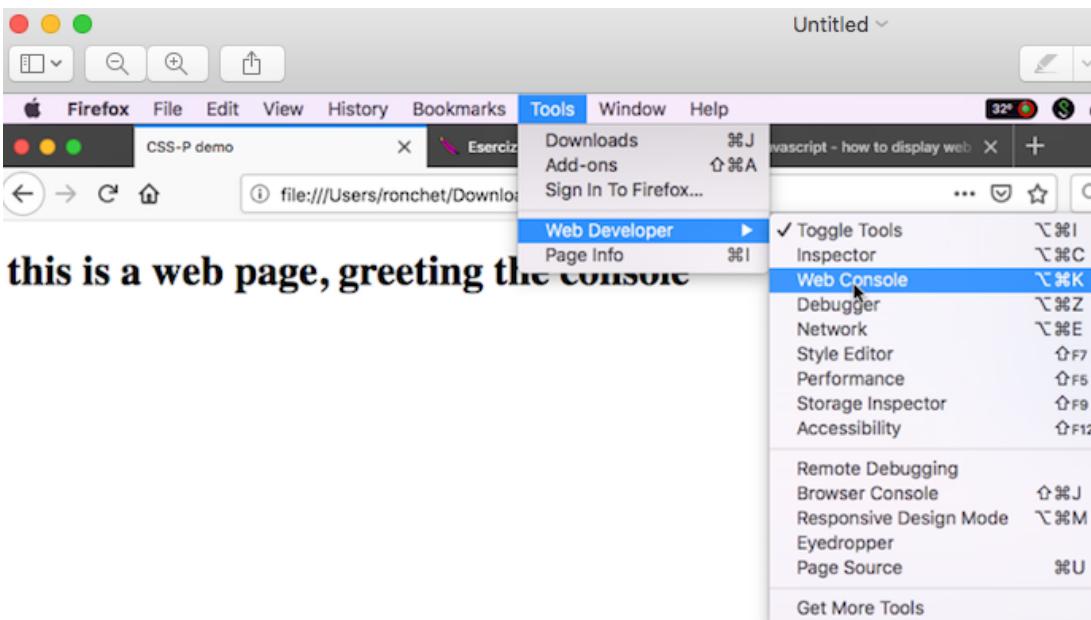
OUTPUT:

1:x in f A  
ERROR:Cannot access 'x' before initialization



# Javascript: the language

## 8 - The JavaScript Console

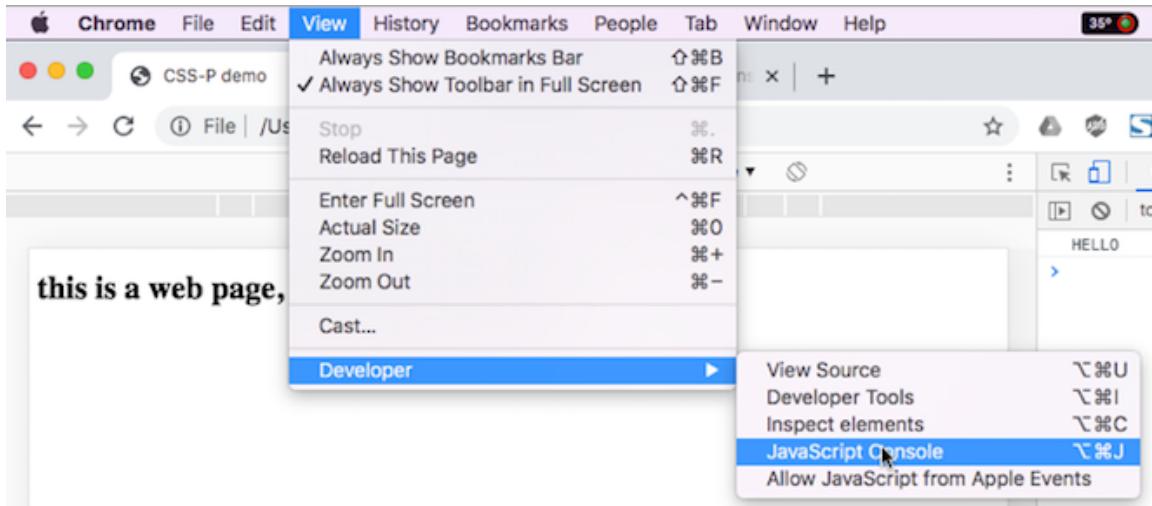


# The Javascript console

Firefox

A screenshot of a Firefox window showing the 'Console' tab of the Web Developer Tools. The tabs at the top are 'Inspector', 'Console' (which is active and highlighted in blue), 'Debugger', 'Network', 'Logs', 'Info', 'Debug', 'CSS', 'XHR', and 'Requests'. The console output shows two entries: 'HELLO' at line 19:17 and another 'HELLO' at line 10:17, both from 'prova.html'. The URL in the address bar is 'file:///Users/ronchet/Downloads/prova.html'. The title bar has three tabs: 'CSS-P demo', 'Esercizio14.3.pdf', and 'javascript - how to display web'. The main content area displays the text 'this is a web page, greeting the console'.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>CSS-P demo</title>
  </head>
  <body>
    <h1>this is a web page, greeting the console
    <script>
      console.log("HELLO")
    </script>
  </body>
</html>
```



```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>CSS-P  
demo</title>
  </head>
  <body>
    <h1>this is a web page,  
greeting the console
    <script>
      console.log("HELLO")
    </script>
  </body>
</html>
```

# The Javascript console

Chrome

Responsive ▾ 686 × 573 100% ▾ Online ▾

Console

top

prova.html:10

HELLO

# Javascript: the language

## 9 - Objects

# Javascript objects

JavaScript is confusing for developers coming from Java or C++, as it's all dynamic, all runtime, and it **has no classes at all**.

It's all just instances (objects).

Even the "classes" we simulate (introduced in ES5) are just a **function** object.

*from developer.mozilla.org*



# Javascript objects as data structures

Javascript objects are collections of **named values (properties)**

```
var person = {firstName:"Dorothea",
lastName:"Wierer", birthyear:1990};
```

Javascript objects can contain also **methods**

```
var person = {firstName:"Dorothea",
lastName:"Wierer", birthyear:1990,
fullName:function() {return this.firstName
+ " " + this.lastName; } };
```



# Accessing properties

`person.firstName`

is equivalent to

`person["firstName"] ;`

and to

`var x="firstName";`

`person[x] ;`



# Dynamic management of objects

You can dynamically add new properties to an existing object by simply giving it a value.

```
var person = {firstName:"Dorothea",
  lastName:"Wierer", birthyear:1990};
person.birthplace="Brunico";
```

You can also delete existing properties.

```
delete person.firstName;
```



# Other ways to create objects

Hence, Javascript objects can also be created empty and populated later.

```
var person = {};  
person.firstName="Dorothea";  
person.lastName:"Wierer";  
person.birthyear:1990;  
person.fullName=function() {  
    return this.firstName + " " +  
this.lastName;};
```



# "Object" is a misleading name!

If you come from a language like Java, where objects are instances of classes, you will notice that the "Object" notion of Javascript is quite different!

For instance:

- you can not change the *structure* of a Java object, but you can change the structure of a JavaScript object!
- you cannot have objects without class in Java, but you do in JavaScript!



# Other ways to create object

```
var object1 = new Object();  
Object.defineProperty(  
    object1, 'name', {  
        value: "AA",  
        writable: false  
});  
object1.name = 77;  
document.write(object1.name);
```

OUTPUT:  
AA

(but no error)

if writable:true:

OUTPUT:  
77



# Objects constructors

Object constructors: templates for creating objects of a certain type (somehow similar to the concept of "class").

```
function Rectangle(w, h) {  
    this.width=w;           ← Instance variables  
    this.height=h;          ←  
    this.area=function(){return this.width*this.height}  
}  
a=new Rectangle(3,4);      ← method  
                           a.area() => 12    a.width => 3
```

*The constructor function is JavaScript's version of a class.*



# Odd consequences...

```
<script>  
p = (n,s) => document.write(n+": "+s+"<br>");  
function Rectangle(w, h) {  
    this.width=w;  
    this.height=h;  
    this.area=function() {  
        return this.width*this.height}  
}  
  
a=new Rectangle(2,3);  
b=new Rectangle(2,3);  
p(a.area(),b.area());  
a.area=function(){return this.width*5}  
p(a.area(),b.area());  
</script>
```

OUTPUT:  
6:6  
10:6



# Using the console

The screenshot shows the Chrome DevTools Sources tab for a file named "prova.html". The code editor displays a script block with a function Person that logs its properties to the console. A tooltip for the "fullname" property is shown, listing its getters and setters. The call stack shows the execution path from the script to the current context.

```
<script>
    function Person(first, last) {
        // property and method definitions
        this.name = {
            'first': fullname,
            'last': lastname
        };
        this.fullname = function () { return this.name.first + this.name.last; }
    }
    let person1 = new Person('Dorothea', 'Wierer');
    console.log(person1);
</script>
```

{} Line 15, Column 12

Console What's New

1 message

1 user messages

No errors

No warnings

1 info

No verbose

The screenshot shows the Chrome DevTools Sources tab for a file named "prova.html". The code editor displays a script block with a Person constructor function that logs its fullname property. The call stack shows the execution path from the script to the current context.

```
<title>CSS-P demo</title>
</head>
<body>
    <h1>this is a web page, greeting the console</h1>
    <script>
        function Person(first, last) {
            // property and method definitions
            this.name = {
                'first': first,
                'last': last
            };
            this.fullname = function () { return this.name.first + this.name.last; }
        }
        let person1 = new Person('Dorothea', 'Wierer');
        person1.name.first = "AAA";
        console.log(person1.fullname());
    </script>
</body>
</html>
```

{} Line 18, Column 20

The screenshot shows the Chrome DevTools Console tab for the file "prova.html". The console output shows the result of logging the fullname property of person1, which is "AAA Wierer".

AAA Wierer

prova.html:20

{} Line 18, Column 20

Console What's New

1 message

1 user messages

No errors

No warnings

1 info

No verbose

# Objects

The approach we have shown is not the most efficient in terms of memory allocation, as for every Rectangle we instantiate the area method!

It would be better to use the “prototype” feature.

```
Rectangle.prototype.area=function () {  
    return this.w*this.h  
}
```

