Javascript

Giuseppe Della Penna
Università degli Studi di L'Aquila

dellapenna@univaq.it
http://www.di.univaq.it/gdellape

Notes to the English Version

These slides contain an English translation of the didactic material used in the Web Engineering course at University of L'Aquila, Italy.

The slides were initially written in Italian, and the current translation is the first result of a long and complex adaptation work.

Therefore, the slides may still contain some errors, typos and poorly readable statements.

I'll do my best to refine the language, but it takes time.

Suggestions are always appreciated!

Using JavaScript

Scripts in HTML pages

- 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.

Scripts in HTML pages

Examples

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

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

Using JavaScript
Running Scripts in HTML pages

- 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.

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 pioint.
  - The *parseFloat* and *parseInt* 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 one value, null.

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.

Variables

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
```

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

Operators

- **&& (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

Operators

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 “ciao”

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.
```

Control Structures

Conditional execution

- JavaScript has an **if** statement with the same syntax as Java:
- **if *(expression) {body}* else *{body-else}***
- Not-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.
- Execute some instructions only if a specified variable or property is defined and not empty, simply write **if (variable) {...}**

Control Structures
Conditional execution

- 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 .

Control Structures

Examples

```
var s = “valore”;
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; //questo case finisce qui
   case “error”: … //questo case continua sul default
   default: …
}
```

Control Structures
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 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.

Control Structures

The for...in loop

- 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.

Control Structures

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
}

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);
```

Functions

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 constituted by 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.

Functions

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;”);
```

Functions

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).

Funcions

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.

Functions

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
```

Functions

Passing Parameters

- Parameters passing inJavaScript 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.

Functions

Returning Values

- 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.

Functions

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).

Functions

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.

Functions

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* function that calls it *closure* with the desired parameters.
- See the following examples ...

Functions

Closures: Examples

```
function f(x) {
   return x+global_variagle; //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) (not the value of 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 balled with a parameter (e.g., we could write o.p(3))

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

Functions

Closures: Examples

```
//we want to assign an handler for the onclick event to an HTML DOM element
htmlelement.onclick = f; //NOTE: we canno 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.
//For example, we want to associate to a set of elements an handler which sets their color to red when they
are clicked
function clickHandler(oToHighlight) {
   return function(e) {
      oToHighlight.style.backgroundColor="red";
   }
}

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

Javascript objects

- **Javascript is an object oriented language,** but his concept of object is much more similar to an associative array.
  - Javascript objects contain methods, which can however be considered property values, as they are merely objects of class **Function.**
- In Javascript it is not possible to define **classes,** but only special special **constructor** functions 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.

Javascript objects

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 (i.e, if you assign a value to an object property, it is automatically created if not already present).
  - It is not possible to add properties to variables of non-object type

Javascript Objects

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
```

Javascript Objects

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}***
- The methods can be added at any time to an object, just as the properties.
- The methods of an object, 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!

Javascript Objects

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 meotodo1

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
```

Javascript Objects

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 operator **new**, just like the standard JavaScript object names:

  new *function(parameters)*
- The constructor functions should never be called directly.

Javascript objects

Constructor Functions

- When using a **new** constructor, 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 may populate the new object, adding properties and methods through **this.**
- Note again that the methods of an object, to refer to the property of the same object, must use the **this** keyword.

Javascript objects

Constructor Functions-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 x
*/

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

o.getW = function() {return this.w;} //dynamic addition of object member (NOT specified in the constructor)
o.getV = function() {return v;} //ERROR! V points to the GLOBAL variable v!
```

Javascript objects

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.
- However, 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 *__proto__* (less supported) property.

Javascript objects

Prototypes-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); //ritorna 3
o2.m(1); //ritorna 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!
```

Javascript objects

Prototypes-Examples

```
//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
```

Javascript objects

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.

Javascript objects

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.

Javascript objects

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);
```

Javascript objects

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*, and not with *this*) will be a private member of the objects it generates.
- Private methods **can access the 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 OO 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!

Javascript objects

Private members-Examples

```
function myObject(a) {
   this.v = a+1; //pubblic 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
```

Javascript objects

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.

Javascript objects

Privileged members-Examples

```
function myObject(a) {
   this.v = a+1; //pubblic 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 members
}
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
```

Predefined Javascript Objects

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

Predefined Javascript Objects

RegExp and String

- JavaScript recognizes 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 **a**rray 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 modifier /s

Predefined Javascript Objects

Arrays

- 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 *variabile_array [index]*
- It is possible to check if an index is present in an array using the boolean expression ***index in*** *variabile_array*.
- 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 data 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 array
  - **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 .

Predefined Javascript Objects

Arrays-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; } //ise an array element only if it is defined

/*
Note: to create an associatve array, you can simply dynamically create new properties (array keys) to an
empty Object
*/
```

Predefined Javascript Objects

Dates

- 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** (set the day of the month: if the passed value is greater than the maximum allowed, *the function automatically increases the month/year)*

Predefined Javascript Objects

Dates-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()] + " " +oggi.getFullYear();

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



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

Predefined Javascript Objects for Browsers

window

- 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 the 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 displayedHTML document.
    - Other properties, such as **statusbar**, have still a very browser-specific semantics.

Predefined Javascript Objects for Browsers

window-Examples

```
//executes an action only if the user clicks OK
if (window.confirm("Sei sicuro di voler lasciare questa pagina?")) {…}

//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!");
```

Predefined Javascript Objects for Browsers document

- The **document** object ca 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.

Predefined Javascript Objects for Browsers document-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();
```

Predefined Javascript Objects for Browsers

XMLHttpRequest

- 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!

Predefined Javascript Objects for Browsers

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();
- In Internet Explorer, you may have 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");

Predefined Javascript Objects for Browsers

XMLHttpRequest: instantiation- Example

```
function createRequest() {
    var ACTIVEXIDs=["MSXML2.XmlHttp.6.0","MSXML2.XmlHttp.3.0"];
     if (typeof XMLHttpRequest != "undefined") {
          return new XMLHttpRequest();
     } else if (typeof ActiveXObject != "undefined") {
        for (var i=0; i < ACTIVEXIDs.length; i++) {
            try {
                return new ActiveXObject(ACTIVEXIDs[i]);
            } catch (oError) {
                //the object does not exist: try the other one
            }
            alert("Impossible to create an XMLHttpRequest");
        }
    } else {
        alert("Impossible to create an XMLHttpRequest");
     }
}
```

Predefined Javascript Objects for Browsers

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 unavalable 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*.

Predefined Javascript Objects for Browsers

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);

- Checks whether the request returned an HTTP error using the ***status*** property, for example

  if (xhr.status! = 404) {...}

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

Predefined Javascript Objects for Browsers

XMLHttpRequest: synchronous use - Example

```
var req = createRequest();

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

req.send(null);

if (req.status!=404) {
   alert(req.responseText);
} else {
   alert("error");
}
```

Predefined Javascript Objects for Browsers

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 the *handler* to call when the request has been served using the ***onreadystatechange*** property:

  xhr.onreadystatechange = function () {...};

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

  xhr.send(null);

- Within the event handler, first verify if the request was served by checking that the ***readyState*** property equals to 4:

  If (xhr.readyState == 4) {...};

- If the request was served, you can check whether the request returned an HTTP error using the ***status*** property and then access the data returned by the server via the ***responseText*** property, as already explained.
- At any time, you can invoke the ***abort*** method to stop the current HTTP request.

Predefined Javascript Objects for Browsers

XMLHttpRequest: asynchronous use - Example

```
var req = createRequest();

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

req.onreadystatechange = function () {
   if (req.readyState==4) {
        if (req.status!=404) {
          alert(req.responseText);
        } else {
          alert("error");
        }
   }
};

req.send(null);
```

Predefined Javascript Objects for Browsers

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)();
```

Exception Handling

- 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.

Exception Handling

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.

Exception Handling

Handlers-Example

```
try {
   ... code…
} catch(e) {
   //exceptions generated by Javascript are object whose message property
   //contains the error message
   alert("Exception raised: "+ e.message);
}
try {…code… } catch (eccezione) {…exception handling…}
finally {
   ...code executed in any case before the execution point goes after the try block...
}
try {
   …code…
   throw {"nome": "pippo", "valore": 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
}
```

Useful JavaScript Functions

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()**

Useful JavaScript Functions

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);
```