# JavaScript

August 30, 2011

## 1 JavaScript Fundamentals

### 1.1 Datatypes

Javascript has three primitive datatypes: **number**, **string** and **boolean**. Javascript also defines two trivial datatypes: **null** and **undefined**. In addition to these primitive datatypes, JavaScript supports a composite datatype denoted by **object**. An object is simply a collection of objects, either primitive values or other objects. Objects in JavaScript have a dual nature: an object can represent an unordered collection of named values or an ordered collection. In the former case, the object is called an **array**. Although objects and arrays are fundamentally the same datatype in Javascript, they behave quite differently. There is another special kind of objects: *functions*. There are still other specialized kinds of objects: **Date**, **RegExp** and **Error**. We will briefly describe how javascript handles each one of these datatypes.

- All *numbers* in JavaScript are represented as floating-point values. When a number gets larger than the largest representable finite number, Javascript uses the constant **Infinity** to represent it. Similarly, when a number becomes lower the last representable negative number, JavaScript uses **-Infinity** to represent it. When numeric operations yield an undefined result or an error a special constant **NaN**, *not a number*, is returned. Complex mathematical operations are stored as properties of a single **Math** object.

- A *string* comprises a sequence of Unicode characters within single or double quotes. Like JavaScript, HTML also allows single and double quotes. Thus, when combining JavaScript and HTML it is a good idea to use one style of quotes to HTML and the other to JavaScript. '+' is used for string concatenation. There are several methods available to operate on strings like: **charAt**, **indexOf** and so on. Also, the lenght of a given string is recorded in the property **length**. There are two main ways to convert a number to a string: the **String** function and the **toString** method of the **Number** object. You can also use one of the three following methods: **toFixed**, **toExponential** and **toPrecision**. When strings are used in numeric contexts they are automatically converted to a number.

- The boolean values are **true** and **false**. The equality testing operator is '=='. If a boolean value is used in a numeric context, true is converted to 1 and false is converted to 0. If a number is used in a boolean context it is converted to 1 except for 0 or **NaN**. One can make this conversion explicit using the **Boolean** function. The Not operator is '!'.

- An important feature of JavaScript is that functions are values that can be manipulated by the JavaScript code. In many languages, including Java, functions are only a syntactic feature of the language: they can be defined and invoked but they are not datatypes. The fact that functions are true values in JavaScript gives a lot of flexibility to the language. It means that functions can be stored in variables, arrays and objects, and it means that functions can be passed as arguments to other functions. Since functions are values just like numbers and strings, they can be assigned to object properties just like any other values. When a function is assigned to a *property* of an object it is called a *method.* There are two ways to define a function literal. The first way is using the **function** keyword:

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

  The second way consists in using the **Function** constructor:

  ```
  var x = new Function("x", "return x*x;");
  ```

- An *object* is just a collection of named values. These named values are usually referred to as *properties* of the object. If an object named image has the properties **width** and **height**, one can refer to these properties in the following way:

  - image.width and image.height
  - image["width"] and image["heigh"]

  Objects are created by invoking special *constructor functions.* Once you create an object you can use and set its properties however you desire:

  ```
    var point = new Object();
    point.x = 2.3;
    point.y = -1.2;
  ```

  JavaScript defines an object literal syntax that allows you to create an object and specify its properties. Object literals can be nested.

  ```
    var point = { x:2.3, y:-1.2};
    var rectangle = { upperleft  : { x: 2, y: 2 },
                      lowerright : { x: 4, y: 4 }
                    };
  ```

Finally, the property values used in object literals need not to be constants, they can be arbitrary JavaScript expressions. Also, the property names in object literals may be strings rather than identifiers.
When a non-null object is used in a Boolean context, it converts to **true**. When an object is used in a string context, JavaScript calls the **toString** method of the object and uses the string value returned by that method. When an object is used in a numeric context, JavaScript calls the **valueOf** method, if it does not return a number, JavaScript first converts the object to a string and then the string to a number.

- An array is a collection of data values, just as an object is. While each data value contained in an object has a *name*, each data value contained in an array has an *index*. In JavaScript, you retrieve a value from an array by enclosing an index in square brackets after the array name. An array can be created with the **Array** constructor function. Once created, any number of indexed elements can be easily assigned to the array:

  ```
  var a = new Array();
  a[0] = 1.2;
  a[1] = ``JavaScript'';
  a[2] = true;
  a[3] = { x: 1, y: 2 };
  ```

  Arrays can also be initialized by passing array elements to the **Array()** constructor:

  ```
  var a = new Array(1.2, ``JavaScript'', true, { x: 1, y: 2 });
  ```

  When only a single number is passed to **Array** then it specifies the size of the array. JavaScript defines a literal syntax for creating and initializing arrays (naturally arrays can be nested within arrays):

  ```
  var a = [1.2, ``JavaScript'', true, { x:1, y:2 }, [1, 2]];
  ```

  Undefined elements can be included in the specification of an array literal just by omitting a value between commas.

- **null** is a special JavaScript value indicating no value. **null** is usually considered a special value of *object* type. When **null** is used in a boolean context it converts to false. When used in a numeric context it converts to 0. When used in a string context it converts to “null”.

- Date and time values are not one of the fundamental datatypes supported by JavaScript. However, JavaScript does provide a class of object that represents dates and times and can be used to manipulate this type of data. A **Date** object is created with the **new** operator and the **Date** constructor.

```
var now = new Date();
var xmas = new Date(2011, 11, 25);
```

Months are zero based so December is month 11.

- *Primitive datatypes wrapper objects.* A corresponding object class is defined for each of the three primitive datatypes. That is, besides supporting the number, string and boolean datatypes, JavaScript also supports **Number**, **String** and **Boolean** classes. These classes are wrappers around the primitive datatypes. A *wrapper* contains the primitive data value, but it also defines properties and methods that can be used to manipulate that data.
  When you use a string in an object context, JavaScript creates a *transient* String object. It allows you to access a property or a method and when it is no longer needed, it is reclaimed by the system. If you want to use a String object explicitly, you have to create it like any other object using the **new** operator.

  ```
  var s = ``hello world'';
  var S = new String(``hello world'');
  ```

  Naturally when a wrapper object String is used in a primitive string context, a transient primitive string is created. Finally, note that any number, string, or boolean can be converted to its corresponding wrapper object with the **Object** function.

In JavaScript *primitive types* are passed by value, whereas *reference types* (objects and arrays) are passed by reference. Note that strings are primitive types so they are passed by value. The question of whether strings are passed by value or by reference is however an empty question because strings are immutable, that is you cannot modify a string (if you want to modify a string you have to create another string). So, the advantages of reference types are not available for strings. Additionally strings are compared by value and not by reference (using '==').

## 1.2 Variables and scope

JavaScript is an *untyped language.* A feature related to JavaScript's lack of typing is that the language conveniently and automatically converts values from one type to another, as necessary. Before you use a variable in JavaScript you may choose to declare it. Variables are declared with the **var** keyword.

```
var i;
var sum;
```

You can declare multiple variables with the same **var** keyword:

```
var i, sum;
```

And you can combine variable declaration and variable initialization:

```
var i = 0, sum = 1;
```

If you don't declare the variable with the **var** statement, the variable is implicitly declared but its initial value is set to **undefined**. Note that the var statement can also appear as part of the **for** or the **for/in** loops.
As referred above, when a variable is not declared with the **var** keyword, it is implicitly declared. However, it is always declared as a global variable even if it appears within the body of a function. Global variables have global *scope*, whereas variables declared in a function (and also the parameters of the function) are only visible within the body of the function. It is important to emphasize that JavaScript does not have *block-level scope*, this means that all variables declared within a function, no matter where they are declared, are defined throughout the function.

```
var scope = ''global'';
function f() {
  alert(scope);
  var scope = ''local'';
  alert(scope);
}
f();
```

One may expect the first call to **alert** to display "global". However, this is not the case since within the body of function f, variable **scope** is understood as a local variable. In the first invocation of alert, **scope** is interpreted as a local variable not yet initialized. Therefore, its evaluations yields the **undefined** value.
It is important to make a distinction between *undeclared and uninitialized* variables and *uninitialized* variables. *Undeclared and uninitialized* variables are variables which are neither declared nor initialized. When attempting to read the value of such a variable JavaScript will generate a runtime error. *Uninitialized* variables are variables which are declared but are not initialized. They are automatically assigned the **undefined** value and therefore, when attempting to read the value of such a variable, you will get the **undefined** value.
Garbage collection is performed automatically in JavaScript.
What is the difference between variables and properties? None. In JavaScript variables and properties are fundamentally the same. When the JavaScript interpreter starts up, one of the first things it does before executing any JavaScript code is to create a *global object*. The properties of this global object are the global variables of the JavaScript program it corresponds to. In top-level code, you can use the keyword **this** to refer to the global object. While the body of a function is executing, the function arguments and local variables are stored as properties of a special object deemed the *call object*. Every JavaScript execution context has a *scope chain* associated with it. This chain is a list of *call objects*. When JavaScript needs to look up the value of a variable x, it starts by looking

at the first object in the scope chain. If that object has a property named x, the value of that property is used. If the first object does not have a property named x, JavaScript proceeds to the next object in the scope chain.

## 1.3 Expressions

We shall now review very quickly the operators which are not so common in other languages:

- The operators == and === check if two values are the same, using two different definitions of equality. First of all, it is important to stress that for both these operators the following holds: primitive types (booleans, strings and numbers) are compared by value whereas objects, arrays and functions are compared by reference. The difference between == and === is that == will test equality after doing any necessary type conversions, whereas if we provide the operator === two arguments of different types, it will return false. Therefore, we shall say that == tests if two values are *equal* and === checks if two values are *identical*.

- The **in** operator expects a left-hand side operand that is or can be converted to a string. It expects a right-side operand that is an object or array. It evaluates to true if the left-side value is the name of a property of the right-side object.

  ```
  var point = { x:1, y:1 };
  var has_x_coord = "x" in point;
  ```

- The **instanceof** operator expects a left-hand side operand that is an object and a right-hand side operand that is the name of a class of objects. The operator evaluates to true if the left-side object is an instance of the right-side class and evaluates to false otherwise.

- In JavaScript the assignment is an expression using the = is an operator. The value of an assigment expression is the value of the right-hand side operand. As a side effect, the = operator assigns the value on the right to the variable, array element, or property. The assignment operator associates to the right, which means that $i = j = k = 0$; is a valid expression.

- The conditional operator is the only ternary operator:

  ```
  x > 0 ? x*y : -x*y
  ```

- The **typeof** operator evaluates to “number”, “string” and “boolean” if its operand is a number, a string, or a boolean respectively. It evaluates to “object” for objects, arrays and null. It evaluates to “function” for functions and it evaluates to “undefined” for the undefined value. Since typeof evaluates to “object” for all objects, it is usually used only to distinguish objects from primitive types.

- The **new** operator creates a new object and invokes a constructor function to initialize it. It has the following syntax:

  ```
  new constructor(arguments)
  ```

  The **new** operator first creates a new object with no properties defined. Then, it invokes the specified constructor function, passing the specified arguments and passing the newly created object as the value of the **this** keyword. The constructor function can then use the **this** keyword to initialize the new object in any way desired.

- The **delete** operator deletes the object property, array element or variable specified as its operand. It returns true if the deletion was successful and false if the operand could not be deleted. Not all variables and properties can be deleted: some built-in core and client-side properties are immune to deletion. Additionally, user defined variables declared with the **var** statement cannot be deleted. If delete is invoked on an non-existant property it returns true.

- The **void** operator discards its value operand and returns **undefined**. It is used in two distinct ways: it allows you to evaluate an expression for its side-effects without the browser displaying the value of the evaluated expression and to purposely generate the **undefined** value in such a way that ensures backward compatibility.

- The , (comma) operator evaluates its left argument and its right argument, and returns the value of its right argument.

- You can access the elements of an object using the dot operator and the elements of an array using the [] operator. Naturally, the . operator expects an object as its left-side operand and an identifier as its right-side operand. If the property specified by the right-side operand does not exist, JavaScript returns undefined. Note that the dot operator is unique since its right-side operand is required to be an identifier (it cannot be an expression that evaluates to an identifier, nor a string, it must be an identifier). The [] operator allows access to array elements and object properties. For accessing array elements its right-side operand must be a number, for accessing object properties its right-side operand must be a string.

  ```
  for (f in o) {
    document.write('o.' + f + '= ' + o[f]);
  }
  ```

## 1.4 Statements

*Expressions* are JavaScript phrases that can be evaluated to yield a value. Operators within an expression may have side effects, but in general expressions

don't do anything. To make something happen you use a JavaScript *statement*. Statements in JavaScript are separated by semicolons.
The simplest kind of statements in JavaScript are expressions which have side effects. When terminated with a semicolon an expression with a side effect is deemed an *expression statement*.
If we enclose an arbitrary number of statements within curly braces, we obtain a *statement block*. The primitive statements within a statement block end in a semicolon, but the block itself does not. Note that JavaScript syntax specifies that compound statements contain a single substatement. Using statement blocks you can place any number of statements within this single allowed substatement.
Bellow we present a very succinct description of each statement:

- **if**. By default an **else** clause is part of the nearest **if** statement. For a sequence of conditional tests one should use the **else if** program construct.
- **switch**. JavaScript allows each case to be followed by an arbitrary expression. The **switch** statement first evaluates the expression that follows the switch keyword and then evaluates the case expressions in the order in which they appear until it finds a value that matches. The matching case is determined using the === identity operator, not the == equality operator, so the expressions must match without any type conversion. The default statement (if there is one) is surprisingly labeled **default**.
- **Loops**: **while**, **do/while**, **for**. These loops work as one expects them to work.
- **for/in**. This statement provides a way to loop through the properties of an object. The syntax is the following:

  ```
  for (variable in object)
    statement
  ```

  variable should either be the name of a variable or a **var** statement declaring a variable, an element of an array or the property of an object. **object** is the name of an object or an expression that evaluates to an object.

  ```
  for(var prop in my_object)
    document.write("name: " + prop + " ;value : " + my_object[prop]);
  ```

  JavaScript does not specify the order in which the properties of an object are assigned to the variable. If the body deletes a property that has not yet been enumerated, that property will not be enumerated. If the body creates a new property that property will be enumerated. Not all properties are enumerated, properties which have been flagged *nonenumerable* are not.

- *Labels*. Any statement may be labeled by preceding it with an identifier and a colon.

  ```
  identfier: statement
  ```

  By labeling a statement you give it a name that you can use to refer to it elsewhere in your program.

- **break**. The **break** statement causes the innermost enclosing loop or switch statement to exit immediately. JavaScript allows the break keyword to be followed by the name of a label:

  ```
  break labelname;
  ```

  The only restriction on the label of the break is that it must name an enclosing statement. You need the labeled form of the break statement only when you are using nested loops or switch statements and need to break out of a statement that is not the innermost one.

- **continue**. The **continue** statement behaves as expected. We stress the fact that the semantics of this statement depends on the type of loop in which it appears.

- The **var** statement creates defines a named variable by creating a property with that name in the call object of the enclosing function or, if the declaration does not appear within a function body, in the global object. Note that enclosing a **var** statement in a **with** statement does not change its behavior. If no initial value is specified for a variable with the **var** statement, the variable is defined, but its initial value is set to **undefined**.

- The **function** statement has the following syntax:

  ```
  function funcname([arg1 [, arg2 [..., argn]  ...]]) {
  }
  ```

  funcname is the name of the function being defined. This must be an identifier, not a string or an expression. The curly braces are a required part of the function statement. Technically speaking, the function statement is not a statement. Statements cause dynamic behavior in a JavaScript program, while function definitions describe the static structure of a program. Statements are executed at runtime, but functions are defined when JavaScript code is compiled.
  Functions may be nested within each other. However, they may not appear inside compound statements.

- **return**. When no value is explicitly returned, the value returned is **undefined**.

- **throw**. Syntax:

  ```
  throw expression;
  ```

  expression may be an object of any type. However, it is usually an object of the class **Error**. When an exception is thrown, the JavaScript interpreter immediately stops normal program execution and jumps to the nearest exception handler.

- A **try/catch/finally** statement simply defines a block of code whose exceptions are to be handled. The **finally** block contains statements that are always executed regardless of what happens in the **try** block. If an exception occurs within the **try** block, control is transfered to the **catch** block and only then it is transfered to the **finally** block. However, if there is no local **catch** block, control is transfered to the **finally** block and then propagates to the closest enclosing **catch** block. Naturally, if the finally block itself transfers control, the pending control transfer is abandoned. For example, if a finally clause throws an exception this new exception will override the previous one.

- The **with** statement is used to temporarily modify the scope chain.

  ```
  with(o)
    statement
  ```

  It adds o to the front of the scope chain, executes statement and restores the scope chain. In practice, you can use the **with** statement to save yourself a lot of typing. For instance:

  ```
  frames[1].document.forms[0].address.value
  ```

  Instead:

  ```
  with(frames[1].document.forms[0]) {
     address.value = ‘‘morada fantastica’’;
  }
  ```

  The with statement is difficult to optimize and must be handled carefully.

## 1.5 Objects and arrays

Objects and arrays are two fundamental datatypes in JavaScript. Objects and arrays differ from primitive datatypes because they represent collections of values instead of single values. An object is a collection of named values and an array is an ordered collection of numbered values.
We recall that an object may be used as an *associative array*, that is a data structure that allows you to dynamically associate arbitrary values with arbitrary strings. However, if you want to access an object property using a string

rather than an identifer you must use the [] operator. The dot notation for accessing properties makes JavaScript objects seem like the static objects of C++ and Java, and they work perfectly in that capacity. But they also have the powerful ability to associate values with arbitrary strings.
All objects inherit from the Object class, so all the properties of the Object class are of particular interest:

- **constructor**. Every object has a constructor property that refers to the constructor function that initializes the object.
- **toString**.
- **valueOf**.
- **hasOwnProperty**. Returns true if the object locally defines a noninherited property with the name specified by the single string argument.
- **propertyIsEnumerable**. Returns true if the object defines a noninherited property with the name specified by the single string argument passed to the method and if that porperty would be enumerated by a **for/in** loop.

An array is an ordered collection of values, each value is called an element and each value has a numeric position in the array which deemed its index. It is important to understand that an array is a special kind of object, but it is indeed an object with a thin layer of extra functionality. Thus, you can define nonnumeric object properties on an array and access them using the `.` or [] syntax. When accessing the elements of an array using the operator [], one can put inside the brackets an arbitrary expression. If the expression inside the brackets does not evaluate to a number, JavaScript will convert it to a string and consider it a normal object property. Hence the following statement does **not** yield an exception:

```
a[2.43] = 1.5;
```

Arrays in JavaScript are *sparse*, that is, they do not have a fixed number of elements. The **delete** operator does not delete the elements of an array; when applied to an element of an array, it simply sets the element to **undefined**.
It is improtant to emphasize that the property **length** is a read/write value. So, you can set the lenght of an array to a given value. If you set it to a value smaller than the current side, the array is truncated; any elements that no longer fit in the array are discarded, and their values are lost. If you make it larger than it currently is, new undefined values are added at the end of the array to increase it to the newly specified size.
You can specify a multidimensional array by placing an array within another.

```
  var table = new Array(10);
  for(int row=0; i<table.length; row++) {
     table[row] = new Array(row);
  }
  for(int row = 0; row < table.lenght; row++) {
    for(int col = 0; col < table[row].length; col++) {
       table[row][col] = row * col;
    }
  }
```

Arrays can be manipulated through several different methods which we briefly describe below:

- **join**. Converts all elements of the array to strings and concatenates them.
- **reverse**. Reverses the order of the elements of an array (it does not create a new array!!!).
- **sort**. Sorts the elements of an array in place. You can pass a comparison function as an argument to **sort**.
- **concat**. Creates and returns a new array that contains the elements of the original array on which **concat** was invoked followed by each of the arguments. If any of these arguments is an array it is flattened (this flattening mechanism is not recursive!).
- **slice**. It returns a slice of the array on which it is invoked. Its arguments specify the initial index and the final index of the orignal array that encompasses the array to be returned.
- **splice**. It is similar to the **slice** method but besides creating a new array, it removes it from the original array.
- **push** and **pop**. Use arrays as stacks. The **push** method appends one or more elements to the end of the array. The **pop** method deletes the last element of the array.
- **unshift** and **pop**. Behave in a similar way to **push** and **pop**, but instead of adding an element to the end of the array, they add it to the beginning. **unshift** adds an element or elements to the beginning of the array, shifts the existing array elements up to higher indexes to make room, and returns the lenght of the array. **shift** removes and returns the first element of the array.

## 1.6 Functions

JavaScript functions may be nested within each other.

```
function hypotenuse(a, b) {
  function square(x) { return x*x; }
  return Math.sqrt(square(a) + square(b));
}
```

Nested functions may only be defined at the top level of the function within which they are nested. That is, they may not be defined inside the body of an if statement or a while loop. Note that this restriction applies only to functions defined with the function statement. Function literal expressions may appear everywhere.
A *function literal* is an expression that defines an unnamed function. Although function literals create unnamed functions, the syntax allows a function name to be optionally specified, which is useful when writing recursive calls themselves. For example:

```
var f = function fact(x) {
         if(x == 0)
          return 1;
         else
          return x * fact(x - 1);
        };
```

This program defines an unnamed function and stores a reference to it in the variable f. It does not store a reference to the function into a variable named fact, but it does allow the body of the function to refer to itself using this name.
Although a JavaScript function is defined with a fixed number of named arguments, it can be passed any number of arguments when it is invoked. When a function is invoked with fewer arguments than those that are declared, the additional arguments are given the undefined value. And what if a function is invoked with more arguments than those that are declared? Within the function body, the identifier **arguments** is a special property that refers to an object known as the *Arguments object*. The Arguments object allows full access to the argument values. The ith argument passed to the function can be accessed by arguments[i]. It is important to emphasize that **arguments** is not really an array: it is an Arguments object. So, one can see it has an object that happens to have some numbered properties. Besides, the **arguments** array and the named arguments are two different ways of referring to the same variable. Consider the following example:

```
  function f(x) {
    print(x);  // Displays the initial value of the argument
    arguments[0] = null; // Changes the array element and also changes x
    print(x); // Displays null
  }
```

Naturally, the Arguments object has the **length** property, but it also has a property named **callee** which refers to the function that is currently being executed. This property can be used to allow unnamed functions to invoke themselves recursively.
When a function has a large number of arguments, instead of expecting the user of your code to memorize the order of the arguments, you can create a function that accepts an object whose properties are the arguments of your function. So, when writing a function with many arguments: use object properties instead.

The semantics of the **this** keyword was already discussed. However, it is important to reinforce that when invoking a function (not a method) the this keyword always refers to the global object, even when a function is defined within another function.
A *constructor* function is a function that initializes the properties of an object and is intended for use with the **new** operator: the new operator creates a new object and then invokes the constructor function, passing the newly created object as the value of the this keyword.
Functions are also objects. So, they have properties. The property **length** indicates the number of arguments that the function expects. You can also define your own properties (this is useful when a function needs to use a variable whose value persists across invocations). JavaScript provides two special methods to invoke functions:

- f.call(o, 1, 2);. Invokes function f on the object o and passes to f the arguments 1 and 2.
- f.apply(o, [1, 2]);.

### 1.6.1 Scoping Revisited

*Functions in JavaScript are lexically rather than dynamically scoped.* This means that they run in the scope in which they are defined, not the scope from which they are executed. When a function is defined, the current scope chain is saved and becomes part of the internal state of the function. At the top level the scope chain simply consists of the global object, and the lexical scoping is not particular relevant. When you define a nested function however, the scope chain includes the containing function. This means that a nested function can access all of the the arguments and local variables of the containing function.
When the JavaScript interpreter invokes a function, it first sets the scope to the scope chain that was in effect when the function was defined. Next it adds a new object known as the *call object* (*activation object*) to the front of the scope chain. The call object is initialized with a property named *arguments* that refers to the Arguments object for the function. Named parameters for the function are added to the call object next. Any local variables declared with the **var** statement are also defined within this object. Note that unlike the **arguments** variable, **this** is a keyword, not a property in the call object.
Consider a function g defined within a function f. When f is invoked, the scope chain consists of the call object for the invocation of f, followed by the global object. g is defined within f so the scope chain is saved as part of the definition of g. When g is invoked (*no matter where*), the scope chain includes three objects: its own call object, the call object of f and the global object.

```
<script type="text/javascript">
function a() {
 var y = "y defined by a";
 function b() {
   var y = "y defined by b";
   c();
 }
 function c() {
   alert(y);
 }
 b();
}
a();
document.write("Scope Games");
</script>
```

This program will display the message: "y defined by a": since JavaScript is statically scoped. When **c** is invoked within function **b**, the call object of **c** is added to the scope chain that was in effect when c was defined (this scope chain contains the global object and the call object of **a**).
When a function is invoked, a *call object* is created for it and placed on top of the scope chain that was in effect when the function was defined. When the function exits, the call object is removed from the scope chain. When no nested functions are involved the scope chain is the only reference to the call object. When the object is removed from the chain there are no more references to it and it ends up being garbaged collected. But nested functions change this picture. If a nested function is created, the definition of that function refers to the call object of the containing function. If the nested function is used only within the outer function, however, the only reference to the nested function is in the call object. When the outer function returns, the nested function refers to the call object and the call object refers to the nested function, but there are no other references to either one and so both objects become available for garbage collection. However, if you save a reference to the nested function in the global scope (you do so by using the nested function as the return value of the outer function or by storing the nested function as a property of some other object), there is an external reference to the nested function and the nested function retains its reference to the call object of the outer function. We emphasize that if a function stores global references to several nested functions, all of them will share the same state (they will share the same scope chain). JavaScript code cannot access the call object in any way.
JavaScript functions are a combination of code to be executed and the scope in which it is to be executed. This combination of code and scope is known as a closure. Below we present several examples on how to use closures.

```
/* Creates the closure */
var uniqueID = function() {
                 var id = 0;
                 return function() {return id++; };
               }
/* Uses the closure to produce unique ids */
var id1 = uniqueID();
```

```
/* This function creates two closures that are stored
   has properties of an object passed as an argument and
   that share the same scope */

function makeProperty(o, name, predicate) {
  var value;

  o["get" + name]= function() { return value; };

  o["set" + name]= function(v) {
                    if(predicate && (!predicate(v))){
                     string s = "cannot write that value in property";
                     throw new Error(s+name);
                    }
                    else value = v;
                   };
}
```

It is very important to understand that JavaScript is lexically scoped. So, in order to capture the scope of a given function we have to define another function within it and store a reference to it outside the containing function. This is what must be done!!! A very serious example is provided below:

```
<script type="text/javascript">

function inspect(inspector, title) {
  var expression, result;
  while(true) {
    var message = "";
   if(title)
    message = title + "\n";
   if(expression)
    message += "\n" + expression + "==>" + result + "\n";
  else expression = "";
    message += "Enter an expression to evaluate: ";
    expression = prompt(message, expression);
    if(!expression) return;
    result = inspector(expression);
  }
}

function factorial(n) {
  var result;
  var n;

  var inspector = function($) {eval($)};
  inspect(inspector, "Entering factorial()");
  while(n>1) {
    result = result * n;
    n = n - 1;
    inspect(inspector, "factorial() loop");
  }

  inspect(inspector, "Exiting factorial()");
  return result;
}
</script>
```

Some considerations: we use **prompt** to interact with the user and we use **eval** to evaluate a JavaScript expression specified as a string.
It was already mentioned that one can use the **Function** constructor to create an anonymous function. It is important to emphasize the following aspects regarding the use of **Function**:

- The **Function** constructor allows JavaScript code to be dynamically created and compiled at runtime.
- Functions created using the **Function** constructor are not lexically scoped. They are always compiled as if they were top-level functions.
- The **Function** constructor parses the funciton body and creates a new function object each time it is called. If the call to the constructor appears within a loop or within a frequently called function, this process can be inefficient. By contrast, a function literal that appears within a loop or function is not recompiled each time it is encountered. Nor is a different function object created each time a function literal is encountered.

## 1.7 Classes, Constructors and Prototypes

JavaScript does not support classes like languages such as Java, C++ and C# do. Still, however, it is possible to define pseudoclasses in JavaScript. The tools for doing this are *constructor functions* and *prototype objects*.
The **new** operator instantiates an empty object. Hence, it must be followed by a function invocation. It creates a new object, with no properties and then invokes the function, passing the new object as the value of the **this** keyword. A function to be used with the new operator is called a *constructor function.* Note that in JavaScript the constructor function is allowed to return an object, if it does, the returned object becomes the value of the new expression (the object that was the value of the this is simply discarded). The following example presents the constructor of an hypothetical Rectangle class.

```
function Rectangle(w, h) {
  this.width = w;
  this.height = h;
  this.area = function() {return this.width * this.height; }
}
```

Observe that this solution is clearly not optimal. The function **area** is replicated in every rectangle object instantiated. However, this function is the same for all rectangles. This problem can be solved using *prototypes*.
Every function has a *prototype* property that is automatically created and initialized when the function is defined. The initial value of the prototype property is an object with a single property. This property is named **constructor** and refers back to the constructor function with which the prototype is associated. This means that each function is the constructor of the elements that it builds...
The **new** operator creates a new object and then invokes the constructor function. However, before invoking the constructor function it sets the prototype of that object. The prototype of an object is the value of the prototype property of the corresponding constructor.
A constructor provides a name for a class of objects and intializes properties that may be different for each instance of the class. The *prototype object* is associated with the constructor, and each object initialized by the constructor inherits exactly the same set of properties from the prototype. This means that the prototype object is an ideal place for methods and other inherited properties. Properties are not copied from the prototype object into new objects, they merely appear as if they were properties of those objects. So, an object inherits properties even if they are added to its prototype after it is created. Inherited properties behave just like normal properties of an object. They are enumerated by the **for/in** loop and can be tested with the **in** operator. You can distinguish them only with the **Object.hasOwnProperty** method.

```
function Rectangle(w, h) {
  this.width = w;
  this.height = h;
}

Rectangle.prototype.area = function() {
                             return this.width * this.height;
                           };
```

When you read property p of an object o, JavaScript first checks if o has a property named p. If it does not, JavaScript checks if the prototype object of o has a property named p. For obvious reasons, when you write the value of a property, JavaScript does not use the prototype object.
Built-in classes, such as String and Date, have prototype objects to, and you can assign values to them. Although for the sake of code reusability it would be better no to do such a thing.
In OOP the members of a class may be of four basic types: instance properties, instance methods, class properties and class methods. Every object has its own copies of its *instance properties*. In JavaScript these properties are initialized by the class constructor. An *instance* method is a method that is invoked on

the objects of the class. In JavaScript instace methods are defined as properties of the *prototype* object corresponding to the class constructor. *Class properties* are properties of the class itself. In JavaScript class properties are defined as properties of the constructor itself (not the prototype object). *Class methods* are methods which are associated with the class itself rather than the instances. These methods are not invoked on a particular class object (they are like global functions in a certain way). Class methods are defined in JavaScript as properties of the constructor function itself (not the prototype object). Below we present a very simple JavaScript class.

```
function Circle(radius) {
  this.r = radius;
}

Circle.PI = 3.14;

Circle.prototype.area = function() {
                          return this.r * this.r * Circle.PI;
                        }
```

It's important to emphasize that in JavaScript when defining instance methods one must explicitly specify the this keyword before using object properties.
There are several methods that are very useful and thus you may want to implement in your classes. We already covered **toString** and **toValue**. Another method that one may want to implement is **equals**, since JavaScript always compares objects by reference. If you want to sort an array of objects you can provide a sorting function with the comparator function, so you may want to implement two distinct functions:

- **compareTo**. It is an instance method, it takes as an argument another instance of the class.
- **compare**. It is a class method, that takes to objects of the class as arguments. Note that the **sort** method expects to receive as an argument the **compare** method and not the **compareTo** method.

JavaScript can also simulate class hierarchies. In order to make one class a subclass of another, you need to make the prototype object of the first one instance of the former. Below, we provide an illustrative example:

```
function Rectangle(w, h) {
  this.width = w;
  this.height = h;
}

 Rectangle.prototype.area = function() {
                              return this.width * this.height;
                             }
```

```
function PositionedRectangle(x, y, w, h) {
  Rectangle.call(this, w, h);
  this.x = x;
  this.y = y;

}

PositionedRectangle.prototype = new Rectangle();
delete PositionedRectangle.prototype.width;
delete PositionedRectangle.prototype.height;

PositionedRectangle.prototype.constructor = PositionedRectangle;

PositionedRectangle.prototype.contains =
                    new function(x, y) {
                       return (x > this.x && x < this.x + this.width) &&
                              (y > this.y && y < this.y + this.height);
                    }
```

So in order to create a subclass of a given class, you have to:

- Invoke the superclass constructor from the subclass constructor.
- Explicitly create the prototype object as an instance of the superclass.
- Explicitly set the constructor property of the prototype object.
- Delete any properties that the superclass constructor created in the prototype object because what's important are the properties that the prototype object inherits from its prototype (the methods of the superclass).

You can simplify the syntax for constructor chaining by adding a property named superclass to the prototype of the subclass.

```
PositionedRectangle.prototype.superclass = Rectangle;
```

The only way to invoke overriden methods is to do it explicitly:

```
//Invoke the superclass toString method
Rectangle.prototype.toString.apply(this);
```

## 1.8 JavaScript as a Prototype based language

In a prototype based language there are no distinctions between classes and objects. Objects are created by *cloning* other objects. In this paradigm the notion of class is replaced by the notion of *prototype*. A prototype is just a regular instance of the type. Several instances of the same type share their prototype. In JavaScript each object maintains a reference to its prototype, so upon its creation, the attributes of the prototype are not copied to the new object, instead the new object has a reference to its prototype through which it can access its properties. Naturally, a prototype may have its own prototype, thus appearing the notion of *prototype chain*. There is no default cloning operation in JavaScript, however we can simply define one:

```
function clone(original) {
  function F() {}
  F.prototype = original;
  return new F();
}
```

## 1.9 Mixing Classes and Prototypes

It is important to emphasize that we can simulate Class based OOP in JavaScript, but JavaScript does not bind us to the rules of object oriented programming. As such, it is possible for a class at runtime to 'borrow' the methods of another class, which can be useful if the borrowed methods are sufficiently generic.

```
  function borrowMethods(borrowFrom, addTo) {
    from = borrowFrom.prototype;
    to = addTo.prototype;

    for(var m in from) {
      if((typeof from[m])!= 'function') continue;
      to[m] = from[m];
    }
  }
```

Using this function, we can for instance simulate multiple inheritance in JavaScript.

```
  function Colored(c) { this.color = c; }
  Colored.prototype.getColor = function() { return this.color; }

  function ColoredRectangle(x, y, w, h, c) {
    this.superclass(x, y, w, h);
    this.color = c;
  }

  ColoredRectangle.prototype = new PositionedRectangle();
  ColoredRectangle.prototype.constructor = ColoredRectangle;
  ColoredRectangle.prototype.superclass = PositionedRectangle;

  borrowMethods(Colored, ColoredRectangle);
```

## 1.10 Runtime Type Identification

JavaScript is loosely typed. However, there are several techniques that can be used to determine the type of an object. The most obvious way to determine the type of an object is to use the **typeof** operator. The **typeof** operator is specially useful to distinguish objects from primitive types. The typeof works in the following way:

- null - 'object'
- undefined - 'undefined'
- number - 'number'
- string - 'string'
- boolean - 'boolean'
- Object - 'object'
- Array - 'object'
- function - 'function'

The **instanceof** operator can be applied to any object (note that functions are objects) to test if a given object is an instance of a given class. An object is not only an instance of its own class but also of any superclass. If you want to test whether an object is an instance of one specific class and not an instance of some subclass, you can check its constructor property.

```
var d = new Date();
var isobject = d instanceof Object;       //evaluates to true
var realobject = d.constructor == Object; //evaluates to false
```

An interesting feature of the default **toString** method is that it reveals some internal type information about built-in objects. The ECMAScript specification requires that this default **toString** method always returns a string of the form [object class], where class is the internal type of the object. Note that this only works for built-in classes.
*Duck typing.* There is an old saying that states: 'If it walks like a duck and quacks like a duck, it's a duck'. So, translating to JavaScript, if an object has all the properties of a given class, then we can consider it a member of that class.

## 1.11 Creating Modules and Namespaces

If you want to write a module of JavaScript code that can be used by any script and can be used with any other module, the most important rule you must follow is to avoid defining global variables. Anytime you define a global variable, you run the risk of having the variable overwritten by another module or by the programmer who is using your module. The solution instead is to define all the methods and properties for your module inside a *namespace* that you create specifically for the module.

JavaScript does *not* have any specific support for namespaces, but JavaScript objects work quite well for this purpose. If you want to create a module of functions for working with JavaScript classes, you do not define these methods in the global object (the global namespace). Instead, you store references for these functions in a specially created object.
*Rules of the thumb*:

- A module should never add more than one symbol to the global namespace.
- If a module adds a symbol to the global namespace, its documentation should clearly state what that symbol is.
- If a module adds a symbol to the global namespace, there should be a clear relationship between the name of that symbol and the file from which the module is loaded.

Generally the name of a module includes all the path.

```
/**
 *  jose/porcaria.js
 **/

var jose;
if(!jose) jose = {};
jose.porcaria = {};
jose.porcaria.define = function() {...};
```

Notice that this code declares the global variable jose before testing if it is already defined. This has to do with the fact that attempting to read an undeclared and uninitialized global variable raises an exception. This is a special behavior of the global object. If you attempt to read a nonexistent property of a namespace object, you simply get the **undefined** value with no exception.
We recall that the Java convention for globally unique package name prefixes: start with an internet domain that you own, reverse it and use it!
A module may be used as a collection of functions, a class etc. However, a module can also be used as code that is only supposed to run one single time. The best way to structure a module of this sort is to put the code inside an anonymous function that is immediately invoked after being defined. This ensures that no global variables are created.
JavaScript does not provide any way to specify that some properties in a namespace are public and that some are not. Normally, programmers should use comments and naming conventions. However, you can also use closures to achieve the same effect.

```
  if(com || jose || fragoso) return; // cuidado!!!!
  com = {};
  com.fragoso = {};
  com.fragoso.jose = {};
```

```
(function(){
   function functionCounter(){
      counter++;
      return counter;
   }

   var counter = 0;

   com.fragoso.jose.functionCounter = functionCounter;

})();
```

## 1.12 Regular Expressions

In JavaScript regular expressions are represented by **RegExp** objects. **RegExp** objects may be created with the **RegExp** constructor, of course, but they are more often created using the literal syntax. Regular expression literals are specified as characters within a pair of slash (/) characters.

```
var pattern1 = /s$/;
var pattern2 = new RegExp("s$");
```

Naturally there are a lot of reserved characters, to refer to these characters within a regular expression one has to use the escaping character: '\'.
Individual characters can be combined into *character classes* by placing them inside square brackets. Thus the regular expression /[*abc*]/ will match a, b or c. We can take the complement of a character class by applying ˆ. So, the regexp /[ˆ*abc*]/ matches any character other than the specified ones. We can use the hiffen (-) to range over sequences of symbols. The regular expression /[$a - zA - Z0 - 9$]/ will match any letter or digit. To summarize:

- [...] - any character between the brackets
- [ˆ...] - any character not between the brackets
- . - any character except for newline or any other unicode line terminator
- \\$w$ - any ASCII word character - [$0 - 9a - zA - Z\_$]
- \\$W$ - any character that is not an ASCII word character
- \\$s$ - any whitespace character
- \\$S$ - any non-white space character
- \\$d$ - any digit
- \\$D$ - any nondigit

To specify repetition:

- $n, m$ - match the previous item at least n times but not more than m times
- $n,$ - match the previous item at least n times
- $n$ - match the previous item exactly n times
- ? - match zero or one occurrence of the previous item
- $+$ - match one or more occurrences of the previous item
- $*$ - match zero or more occurrences of the previous item

Regular expressions match as many times as possible while still allowing any following parts of the regular expression to match. We say that this repetition is 'greedy'. For example, the regexpression $/a+/$ when applied to *aaa* will match all three occurrences of *a*. To avoid this, simply follow the repetion character or characters with a question mark: ?. Therefore, if we apply $/a{+}?/$ to the string *aaa*, it will match only the first a. Note that *greedy* expression matching works by finding the first position in the string at which a match is possible. For example, the pattern $/a{*}?b/$ when applied to the string *aaab* will match only the first a.
As usual the | character specifies alternatives. Note that alternatives are considered from left to right until a match is found. If the left alternative matches, the right alternative is ignored, even if it would produce a 'better' match. For example, $/a|ab/$ will always match the first alternative.
Parentheses have several purposes in regular expressions. One purpose is to group separate items into a single subexpression so that the items can be treated as a single unit by the operators already introduced. For example: $/Java(Script)?/$. Another purpose of parentheses is to define subpatterns within the complete pattern. For instance, consider the regexp $/[a-z]+\backslash d+/$. When written as: $/[a-z]+(\backslash d+)/$, it is possible to extract the digits from any matches it finds. Additonally, it is possible to refer back to a parenthesized subexpression later in the regular expression. $\backslash n$ matches the same characters that were matched when group number n was first matched (where groups are subexpressions within parenthesis). Group numbers are assigned by counting left parentheses from left to right. Groups formed with **(:?** are **not** numbered. For example, the regexp:

```
/(['"])[^'"]*\1/
```

matches any well formed quote.
Note that when you want to match a certain pattern against a regular expression, you will generally want to operate on the substring that results from the match. So the first index of the match is quite important. For instance:

```
/\sJava\s/
```

This expression matches any occurrence of the word Java when it is between two spaces. However, the resulting substring will begin with the matched space. Moreover, if the word Java appears in the beginning of the sentence it will not be matched. JavaScript provides a set of *regular expression anchors* that allow you to overcome this problem.

- $\backslash b$ - match a word boundary.
- $\backslash B$ - math a position that is not a word boundary.
- ˆ - match the begining of a string or the beginning of a line.
- $ - match the end of a string or the end of a line.
- $? = p$ - a positive lookahead assertion. Require that the following characters, match the pattern p, but do not include those characters on the match.
- $!?p)$ - a negative lookahead assertion.

Example:

```
/[Jj]ava([Ss]ript)?(?=\:)/
```

There is one final element of interest in the regular-expression grammar. Regular expression *flags* specify high-level pattern-matching rules. Unlike the rest of regular expression rules they are specified outside the / characters. Example: $/\backslash bjava\backslash b/gi$.

- i - perform case sensitive matching
- g - perform global matching
- m - multiline mode

There are several string methods to handle regular expression.

- **search**. Receives as an argument a regexp and returns the index of the first match.
- **replace**. Performs a regular search and replace. You can use the contents of a parenthesized subexpression in the replacement string.

  ```
  var quote = /"([^"])"/g;
  text.replace(quote, "‘‘$1’’");
  ```

- **match**. Takes a regular expression as its only argument and returns an array containing the matches. If the regular expression does not have a *g* flag, **match** does not do a global search, it simply searches for the first match. However, **match** returns an array even when it does not perform a global search. In this case, the first element of the array is the matching string, and any remaining elements correspond to the parenthesized subexpressions of the regular expression. The result of the **match** function comes with two special properties: **index** and **input**. The index property contains the index of the first match and the input property contains the original string.

- **split**. Breaks the string on which it is called using the argument as a separator.

  ```
  text.split(/\s*,\s*/);
  ```

We have already mentioned that regular expressions are instances of the class **RegExp**. So you can dynamically construct a regular expression using the constructor **RegExp**. It takes two arguments: the first is a string with the regular expression and the second is a string with the flags. Note that the escaping character for string specification is the same for regular expression specification (the backslash), so if you want to specify a backslash inside a string, you have to use two of them.

```
var zipcode = new RegExp("\\d{5}", "g");
```

Each RegExp object has five properties. The properties **global**, **ignoreCase** and **multiline** correspond respectively to the flags: **g**, **i** and **m**. The property **source** is a readonly string that contains the regular expression. The property **lastIndex** contains the index of the string at which the search for the pattern begins. You can invoke two methods on a regular expression: **exec** and **test**. The former just checks if there is a match and returns a boolean. **exec** behaves like **match**, however it will only identify the first match. However, it will update the **lastIndex** property and the new search will resume from there.

```
var pattern = /Java/g;
var text = "JavaScript is more fun than Java!";
var result;
while((result = pattern.exec(text))!= null) {
   alert("Matched ‘" + result[0] + "’" +
     " at position " + result.index +
     " next search begins at " + pattern.lastIndex);
}
```

Note that **lastIndex** must be explicitly set to 0, if you start a new search before you finish another one (using the same RegExp object).

# 2 Client Side JavaScript

## 2.1 Basics

### 2.1.1 The window object and its properties

The primary task of a web browser is to display HTML documents in a window. In client-side JavaScript, the *Document object* represents an HTML document,

and the *Window object* represents the browser window (or frame) that displays the document. While the *Document* and *Window* objects are both for important for client-side JavaScript, the Window object is more important for one substantial reason: the Window object is the global object in client-side programming.
The Window object represents a web browser window or a frame within a window. In client-side JavaScript, top-level windows and frames are essentially equivalent. It is however possible to write applications that use multiple windows (or frames). Each window involved in an application has a unique Window object and defines a unique execution context for client-side JavaScript code.
Since the Window object is the global object, all global variables are defined as properties of the window object. The Window object defines a number of properties and methods that allow you to manipulate the web browser window. The **document** property refers to the Document object associated with the window, the **window** and **self** properties refer to the Window object itself and the **location** property refers to the Location object associated with the window.
When a web browser displays a framed document, the **frames**[] array of the top-level Window object contains references to the Window objects that represent the frames. Thus, in client-side JavaScript, the expression **document** refers to the Document object of the current window; the expression $frames[1].document$ refers to the Document object of the second frame of the current window.

### 2.1.2 Unobstrusive JavaScript

*What is JavaScript for?* This is a very important question. So, HTML allow us to specify the content of web page, CSS allow us to specify the presentation and JavaScript adds behavior. What kind of behavior? Examples:

- Visual effects like image rollovers
- Sorting the collumns of a table
- Hiding certain content and revealing details selectively
- etc

*Unobstrusive JavaScript* is an important paradigm concerning JavaScript. The idea is that JavaScript should not draw attention to itself, it should not obtrude. The first rule to write unobstrusive JavaScript is to keep JavaScript code separate from HTML markup. To achieve this, you should put all your JavaScript code in external files and include those files into your HTML pages with the **script** tag (setting the **src** attribute of this element to the path of the file in which the code is stored). The second rule is that you must *degrade gracefully.* This means that JavaScript should be primarily used for enhancements to HTML content, but the core content should still be available without JavaScript. The third goal states that JavaScript code must not degrade the *accessibility* of the HTML page.

### 2.1.3 Embedding JavaScript

JavaScript can be applied to an HTML page in several ways:

- *Embedded JavaScript:* specify the JavaScript code between **script** tags.
- *External JavaScript:* use the **script** element specifying the attribute **src**.
- In an event handler you can specify JavaScript code as the value of an HTML attribute such as **onclick**.

If you are using XHTML you should specify your code within a CDATA section (to avoid interpreting JavaScript codes as XML markup):

```
<script><![CDATA[
]]></script>
```

Characters like < and & are illegal XML characters:

- < is interpreted as the start of a new element
- & is interpreted as the start of a character entity

A single HTML file may contain any number of **script** elements. Multiple **script** elements are executed in the order in which they appear within the document. While separate scripts within a single file are executed at different times during the loading and parsing of the HTML file, they constitute part of the same JavaScript program: functions and variables defined in one script are available to all scripts that follow in the same file.

```
<html>
 <head>
   <title>Today's Date</title>
   <script type="text/javascript">
     function printDate() {
       var d = new Date();
       document.write(d.toLocaleString());
     }
   </script>
 </head>
 <body>
   <scipt language="JavaScript">printDate()</script>
 </body>
</html>
```

The **script** tag supports the **src** attribute that specifies the URL of the file containing the script code. A JavaScript file has a **.js** extension. A **script** tag that specifies a **src** attribute behaves exactly as if the contents of the file appeared between script tags. We stress that the closing tag is always necessary. There are several advantages in storing your JavaScripts programs in external files:

- Simplify the HTML document
- Code reusability
- Import code from an arbitrary URL
- If a JavaScript file is shared between several pages the browser will probably cache it, thus allowing it to be loaded more quickly

You should always specify what is the language of the script you are including. If your scripts are all written in JavaScript you can specify exactly that in the following way:

```
<meta http-equiv="Content-Script-Type" content="text/javascript">
```

You can use the **defer** attribute in the **script** element to specify that the evaluation of the corresponding script may be postponed. This is particularly useful when including external scripts.

```
<script defer="defer"> </script>
```

HTML provides a **noscript** element which is only rendered if JavaScript is disabled. As already said, ideally pages should *degrade gracefully*, so this is a useful tool.
Another way to include JavaScript code is to place it in a URL:

```
javascript: var now = Date(); "<h1>The time is:</h1>" + now;
```

JavaScript URLs may also contain JavaScript code statements that perform actions but return no value. In this case the browser will not modify the current displayed document. Hence, to avoid changing the document it is a good practice to add the instruction *void*0; to the end of the JavaScript URL.
The JavaScript pseudoprotocol can be used with HTML attributes whose value should be a URL. Such as: the **href** attribute of a hyperlink and the **action** attribute of a form. You can obviously *bookmark* a JavaScript URL, thus obtaining a *bookmarklet*.

### 2.1.4 Events

The web browser notifies a program of a certain input by generating an *event*. There are several types of events. When an event occurs the web browser tries to invoke an appropriate *event handler* function to respond to the event. Thus, to write dynamic, interactive client-side JavaScript programs, you must define appropriate event handlers and register them with the system, so that the browser can invoke them at appropriate times.
While you can include any number of JavaScript statements within an event-handler definition, a common technique is to simply use event handler attributes to invoke functions that are defined elsewhere within **script** tags.
Theser are the most common event handler:

- **onclick:** This hanlder is supported by all button-like form elements, as well as the anchor element. It is triggered when the user clicks on the element.
- **onmousedown, onmouseup:** These two event handlers are similar to the previous one, but they are triggered separetely when the user presses and releases a mouse button, respectively.
- **onmouseover, onmouseout:** These two event handlers are triggered when the mouse pointer moves over or out of a document element respectively.
- **onchange:** This event handler is support by the input, select and textarea elements. It is triggered when the use changes the value displayed by the element and then tabs or otherwise moves focus out of the element.
- **onload:** this event handler may appear on the body tag and is triggered when the document and its external content are fully loaded. The onload handler is often use to trigger code that manipulates the document content because it indicates that the document has reached a stable state and is safe to modify.

### 2.1.5 Execution of client side JavaScript programs

Scripts are evaluated by the browser in the order in which they appear (except those with the **defer** attribute). Scripts that are placed in the head of the HTML document generally define functions and variables to be used to other scripts (all the scripts that are included in an HTML document are part of the same program). Scripts placed within the body of an HTML document may call the **document.write()** fucntion in order to help building the document. If it does not use the **document.write()** function then it should be put on the header. Note that the execution of scripts that interact with the document elements (through the DOM) should be postponed to after the document is loaded because only then it is sure that all the elements of the document are availabe. In order to invoke this kind of script you should use the attribute **onload** of the body element. Because **onload** event handlers are only invoked when the document is fully parsed, they must not call **document.write()**. If an event handler call **document.write()** on the document on which it is a part, it will overwrite that document and begin a new one. So, event handler should never call this method.
If a document includes a lot of images (it takes a very long time to load), the **onload** event handler may take a long time to be triggered. In this situations you can choose to put the scripts that manipulate the document at the end of the **body** element. This solution does not gather consensus. Another problem with documents that take a very long time to load is that events may be triggered while the document is still loading (and the functions that process these events may not have been processed).

When the user navigates away from the page the **unload** event handler is triggered. This event handler was conceived to allow the programmer to perform clean up operations (for example, if your application opens a secondary window, it gives the user an oportunity to close it).
The Window object acts as the global object. All functions defined in script tags are properties of this object. However, whenever a new document is loaded, the Window object for that Window is restored to its initial state. The properties of a Window object have the same lifetime as the document that contains the JavaScript code that defined those properties. Nevertheless, the Window object itself as a longer lifespan, it exists as long as the window it represents exists. A reference to a Window object remains valid regardless of how many webpages the window loads and unloads.
JavaScript does not support concurrencey. Thus, you should not write scrips and event handlers that are very time consuming because they will introduce a delay into document loading. If you do choose to write sripts that are time consuming you should notify the user so that he does not think that the browser crashed.

### 2.1.6 Compatibilities etc

If you are worried about compatibilities just be proactive - test if the function you are about to use is defined.

```
if(element.addEventListener) {
   element.addEventListener("keydown", handler, false);
}
else {
   alert("xiribitatata");
}
```

This technique is called *feature testing.* Note the browser identifies itself in the HTTP request, so the server side program can decide which HTML page to send depending on the type of browser. The browser type is identified in the **User-Agent** header field. Moreover, the **navigator** property of the Window object also identifies the current browser.

### 2.1.7 IE conditional comments

They are quite useful.

```
<!--[if IE]>
 Isto so vai aparecer no IE
<![endif]-->

<!--[if gte IE 6]>
  Isto so vai aparecer no IE 6 ou para cima upa upa
<![endif]-->
```

IE also supports JavaScript conditional comments. The comment delimeters are $/*@$ and $@*/$. The **cc_on** stands for conditional compilation. You can intersperse JavaScript conditional comments with regular JavaScript comments to produce code that will be fuctional in all browsers. Figure this code out, it is easy!!!!

```
/*@cc_on
  @if (@_jscript)
     alert('you are in IE');
  @else*/
     alert('you are not in IE');
  /*@end
  @*/
```

### 2.1.8 Security

JavaScript's first line of defense against malicious code is that the language simply does not support certain capabilities. For example, client-side JavaScript does not provide any way to read, write, or delete files or directories on the client computer.
JavaScript can script the HTTP protocol to exange data with web servers and it can even download data from FTP and other servers. But JavaScript does not provide general networking primitives and cannot open a socket to, or accept a connection from, another host.
Security features of JavaScript:

- A JavaScript program can open new browser windows, but to prevent pop-up advertisers, many browsers restrict this feature so that it can happen only in response to a user-initiated event such as a mouse click.
- A JavaScript program can close browser windows that it opened itself, but it is not allowed to close other windows without user confirmation.
- A JavaScript cannot obscure the destination of a link by setting the status line text when the mouse moves over the link.
- A script cannot open a window that is to small. Addtionally, a script cannot create a window without a titlebar or status line.
- The value property of HTML file upload elements cannot be set.
- A script cannot read the content of documents loaded from different servers than the document that contains the script. Similarly, a script cannot register event listeners on documents from different servers. This prevents the script from snooping on the user's input. This restriction is called *same-origin policy.*

The origin of a document is defined as the protocol, host and port of the URL from which the document was loaded. Documents loaded from different web

servers have different origins. Documents loaded through different ports of the same host have different origins. So, the *same-origin policy* states that a script can only access a document which was loaded from the same origin of the document in which the script is included. So, if the script opens a document from a different origin, it cannot access it!!!! Additionally, the same-origin policy also applies when scripting http with the XMLHttpRequest object. This object allows client side JavaScript code to make arbitrary HTTP requests but only to the web server from which the containing document was loaded.
In some circumstances the same origin policy is too restrictive. It poses particular problems for large web sites that use more than one server. To support large web sites of this sort you can use the domain property of the Document object. By default the **domain** property contains the hostname of the server from which the document was loaded. You can set this property but only to a string that is a valid substring of the original one. For instance you can set 'jose.santos.com' to 'santos.com'. The domain value must have at least one dot in it. If two windows contain scripts with the same domain value, the same origin policy is relaxed for these two windows and each window can interact with the other.
*Cross site scripting* is a term for a category of security issues in which an attacker injects HTML tags or scripts into a target web site. In general, the way to prevent XSS attacks is to remove HTML tags from any untrusted data before using it to create dynamic document content.
*Denial of service* attacks consist may be achieved in several different ways. For instance, a site can tie up your browser to an infinite loop or alert() dialog boxes, a malicious site can also attempt to tie up you CPU with an infinite loop of meaningless computation (this can be quite undetectable if it uses techniques such as **window.setInterval()**.

## 2.2 Scripting HTTP

It is possible for JavaScript code to script HTTP. HTTP requests can be initiated when a script sets the **location** property of a window object or calls the **submit()** method on a form object. In both cases, the browser loads a new page into the window, overwriting any script that was running there. However, JavaScript can also communicate with a WebServer without causing the browser to reload the currently displayed web page.
The **img**, **iframe** and **script** tags have **src** properties. When the script sets this properties to a URL, an HTTP GET request is initiated to download the content of the URL. A script can therefore pass information to a web server by encoding that information into a query-string portion of the URL. Althought it is possible to do HTTP scripting with these elements, the generated HTTP requests will always be sent to the server via the GET method. The **XMLHttpRequest** is a way more flexible way to do HTTP scripting. Despite its name the **XMLHttpRequest** object can fectch any kind of text document, not only XML documents.
Scripting HTTP with XMLHttpRequest is a three-part process:

- Creating the XMLHttpRequest object
- Specifying and submitting your HTTP request to a web server.
- Synchronously or asynchronously retrieving the server's response.

### 2.2.1 Creating an XMLHttpRequest

IE does not create the XMLHttpRequest in the same way as the other browsers so we have to be especially careful.

```
HTTP._factories = [
         function() { return new XMLHttpRequest(); },
         function() { return new ActiveXObject("Msxml2.XMLHTTP"); },
         function() { return new ActiveXObject("Microso
   ];

HTTP._factory = null;

HTTP.newRequest = function() {
  if(HTTP._factory != null) return HTTP._factory();
  for(i=0; i<3; i++) {
     try {
       var factory = HTTP._factories[i];
       request = factory();
       if(request != null) {
         HTTP._factory = factory;
         return request;
       }
     }
     catch(e) {
       continue;
     }
  }
  HTTP._factory = function() {
           throw new Error("XMLHttpRequest not supported");
      }
  HTTP._factory();
}
```

### 2.2.2 Submitting a Request

After creating the XMLHttpRequest object, the next step is to submit a request to a web server. This is a multistep process:

- First, call the **open()** method to specify the URL you are requesting and the HTTP method of the request. You can specify three different HTTP methods: GET, POST and HEAD. The HEAD method asks the server to just return the headers associated with the URL (this allows the script to check the modification date of a document, without downloading all the document). The third argument specifies if the request is synchronous or asynchronous.

```
request.open("GET", url, true);
```

- The second step to submitting a request consists in setting the request headers:

  ```
  request.setRequestHeader("User-Agent", "XMLHttpRequest");
  request.setRequestHeader("Accept-Language", "en");
  request.setRequestHeader("If-Modified-Since", lastRequestTime.toString());
  ```

- After creating the request object and setting the headers you can submit the request:

  ```
  request.send(null);
  ```

  The argument passed to send should be the body of the request. If you are using the GET method, it shoudl be null, otherwise it should contain the data you want to send to the server (typically form data).

If you specified a synchronous request, the send method is also synchronous. So, it blocks and does not return until the server's response has arrived. It does not return a status code, but you check the status code using the **status** property of the request object. 200 means that the request was successfull and 404 corresponds to a 'not found' error. The request object makes the response of the server available as a string through the **responseText** object. If the response of the server is an XML document, you can also access the document as a DOM document object through the **responseXML** property. In addition to the *status codes* and the *response text* or *document*, the request object also provides access to the HTTP header returned by the server. **getAllResponseHeaders()** returns the response headers as an unparsed block of text, and **getResponseHeader()** returns the value the header whose name is specified as an argument. It is important to emphasize that there is no way to specify a time limit for a XMLHttpRequest, so if the server does not respond you blocked and there is nothing you can do about it (aside from exiting the browser).
To specify an assynchronous JavaScript request you must set the third parameter of open to true. The callback that handles the event corresponding to the response of the server is stored in the property (of the request) **onreadystatechange**. This function is invoked whenever the value of the **readyState** property changes. The **readyState** property corresponds to an integer that specifies the status of the HTTP request:

- 0 - **open()** has not been called.

- 1 - **open()** has been called, but **send()** has not been called.
- 2 - **send()** has been called, but the server has not responded yet.
- 3 - data is being received from the server.
- 4 - the server's response is complete.

The same function handles all of this changes to the **readyState** property. So, you must check the value of the **readyState** property of the request object within the callback. So, when defining the function you must be sure that request object is in the scope of the function (this function is indeed a closure!!!!).

```
var request = HTTP.newRequest();
request.onreadystatechange = function() {
             if(request.readyState == 4) {
                if(request.status == 200) {
                  alert(request.responseText);
                }
             }
         }
request.open("GET", url);
request.send(null);
```

As was already mentioned you can only send a XMLHttpRequest to the server from which the document that uses it was downloaded. Another important restriction is that the XMLHttpRequest makes HTTP requests. It cannot work with URLs that use the **file://** protocol. This means that you cannot test XMLHttpRequests from your local filesystem. You must upload your scripts to a webserver, or run a server in your desktop.
Useful code:

```
HTTP.getText = function(url, callback) {
     var request = HTTP.newRequest();
     request.onreadystatechange = function() {
            if(request.readyState == 4 && request.status == 200)
              callback(request.responseText);
         }
     request.open("GET", url);
     request.send(null);
  }
```

There are several situations in which it's better to use the POST method than the GET method. For instance, if the request has side effects on the server - the same request may generate different responses - using the GET method may cause us to obtain an old response that is cached somewhere and that is no longer valid. When using the POST method the arguments of the request are passed as a string to the **send()** method. The next procedure aims at constructing this string:

```
HTTP.encodeFormData = function(data) {
   var pairs = [];
   var regexp = /%20/g;

   for(var in data) {
     var value = data[name].toString();
     var pair = encodeURIComponent(name).replace(regexp, "+") + ‘=’ +
         encodeURIComponent(value).replace(regexp, "+");
     pairs.push(pair);
   }

   return pairs.join(‘&’);
}
```

The global function **encodeURIComponent** encodes spaces as ‘%20’, so we must fix its output.

### 2.2.3 The server's response

The server response may be a plain text value (**type = ‘text/plain’** ). However it can also be an XML document (**type = ‘text/xml’**), or an HTML document. Naturally, you can treat all kinds of responses as plain text. If the server sends a response of type ‘text/xml’, it will be stored in the **responseXML** property and you can search and traverse it using DOM methods. Note that using XML as a data format may not always be the best choice. If the server wants to be manipulated by a JavaScript script, it is inefficient. A more efficient way is to have the server encode the data using JavaScript object and array literals and pass the JavaScript source text to the web browser. The script then parses the response simply by passing it to the JavaScript eval() method. Encoding data in the form of JavaScript object and array literals is known as **JSON**, or JavaScript Object Notation. The following example illustrates an XML and a JSON encoding of the same data:

```
<!-- XML encoding -->
<author>
   <name>Wendell Berry</name>
   <books>
      <book>The Unsettling of America</book>
      <book>What are people for?</book>
   </books>
</author>

//JSON encoding
{
   "name": "Wendell Berry",
   "books": [
           "The Unsettling of America",
           "What are people for?"
          ]
}
```

The question is: how to process these two different types of response.

```
HTTP._getResponse = function(request) {
   switch(request.getResponseHeader("Content-Type")) {
      case "text/xml": return request.responseXML;
      case "text/json":
      case "text/javascript":
      case "application/x-javascript":
          return eval(request.responseText);
      default: return request.responseText;
   }
}
```

JavaScript does not provide any method to set a timeout on a XMLHttpRequest. However, you can do it yourself with the **window.setTimeout()** function. Normally, you will get your response before the timeout, in this case you just invoke the **window.clearTimeout()**. On the other hand, if your timeout is triggered before the response arrives (**readyState** 4), you can cancel the request using the **abort** method.

```
  HTTP.get = function(url, callback, options) {
     var request = HTTP.newRequest();
     var n = 0;
     var timer;
     if(options.timeout)
       timer = setTimeout(
            function(){
             request.abort();
             if(options.timeoutHandler)
                options.timeoutHandler(url);
            }, otptions.timeout);
    request.onreadystatechange =
      function() {
        if(request.readyState == 4) {
          if(timer) clearTimeout(timer);
          if(request.status == 200) {
             callback(HTTP._getResponse(request));
          }
          else {
            if(options.errorHandler)
               options.errorHandler(request.status,
                                    request.statusText);
            else callback(null);
          }
        }
        else if(otpions.progressHandler) {
           options.pregressHandler(++n);
        }
      }
    var target = url;
    if(options.parameters)
       target += "?" + HTTP.encodeFormData(options.parameters);
    request.open("GET", target);
    request.send(null);
  }
```

Problems with Ajax:

- Feedback on latency
- URLs do not encapsulate state no more
- The Back button does not work

## 2.3 Scripting Browser Windows

JavaScript provides several methods to schedule code to be executed in the future:

- The **setTimeOut()** method of the Window object schedules a fucntion to run after a specified number of miliseconds. This function receives as argument the function to be invoked. If you register a function to be invoked after 0 seconds, you are actually registering it to be invoked as soon as possible (the browser will invoke the function when it has finished running the event handlers for any currently pending events and has finished updating the current state of the document.
- The **setInterval()** method is like the **setTime()** method except that the specified function is invoked repeatedly at intervals of the specified number of milliseconds.
- In order to cancel the execution of a scheduled function you can use **clear-Interval()** and **clearInterval()**.

The **location** object of a window (or frame) is a reference to a *Location object*; it represents the URL of the document currently being displayed in the window. The **href** property of the location object is a string that contains the complete text of the URL. The **toString** method of the Location object returns the value of the **href** property, so you can use **location** in place of **location.href**. Other properties of the **location** object are: **host**, **protocol**, **pathname** and **search**. All these properties are quite obvious except for **search** which contains

the portion of the URL that comes after a question mark. This portion generally corresponds to a list of arguments.
Although the **location** property of a window refers to a *Location* object, you can set it to a string. The browser will interpret this string as a URL and try to display the content of that URL (relative URLs are intepreted with respect to the page in which they appear). You can also use the **replace()** method to display a new web page. However, invoking this method is not equal to changing the **location** property of the window object. When you call **replace()**, the specified URL replaces the current one in the browser's history list rather than creating a new entry in that history list. Therefore, if you use **replace()** the previous page will not be available through the **Back** button. For web sites that use frames and display a lot of temporary pages, **replace()** is often very useful. Do not confuse the **location** property of the Window object with the location property of the Document object. The former is a Location object, whereas the latter is a read-only string. **document.location** is a synonym for **document.URL**. In most cases **document.location** is the same as **location.href**. When there is a server redirect however, **document.location** contains the URL as loaded and **location.href** contains the URL as originally requested.
The **history** property of the Window object refers to the *History* object for the window. The History object is an array of recently visited URLs. You cannot access these URLs, but you have access to the following three methods:

- **back():** move back in the window or frame browsing history.
- **forward():** move forward in the window or frame browsing history.
- **go():** takes an integer argument and can skip any number of pages forward or backward.

### 2.3.1 Window Geometry

Most browsers support a simple set of properties on the Window object that contains information about the window's size and position:

```
//The overall size of the browse window on the desktop
var windowHeight = window.outerHeight;
var windowWidth = window.outerWidth;

//The position of the browser window on the desktop
var windowX = window.screenX;
var windowY = window.screenY;

//The size of the viewport in which the HTML document is displayed
var viewportWidth = window.innerWidth;
var viewportHeight = window.innerHeight;

//The horizontal and vertical scrollbar positions
```

```
//What part of the document appears in the upper left corner of the screen
var horizontalScroll = window.pageXOffset;
var verticalScroll = window.pageYOffset;
```

All these properties are read-only. There are several different coordinate systems you must be aware of. *Screen coordinates* describe the position of a browser window on a desktop; they are measured relative to the upper-left corner of the desktop. *Window coordinates* describe a position within the web browser's view port; they are measured with respect to the upper-left corner of the viewport. *Document coordinates* describe a position within an HTML document. Naturally, when a document is larger or longer than the viewport window coordinates do not coincide with document coordinates. IE behaves differently from the other browser (check the documentation).
The screen property of the Window object is a *Screen object* that provides information about the size of the user's display and the number of colors available on it. The width and height properties specify the size of the display in pixels. These properties can be used to decide whether to include or not certain images. The **availWidth** and **availHeight** the display size that is actualy available.
The **navigator** property of a window refers to a Navigator object that contains information about the Web browser as a whole, such as the version and a list of data formats it can display. You can use this property to perform *browser sniffing*. However, you should not do *browser sniffing*. Instead, you should perform *capability testing* - before using a function check if it exists. The Navigator object has five properties that provide information about the browser:

- **appName:** the name of the web browser.
- **appVersion:** the version of the web browser.
- **userAgent:** the string that the browser sends in its USER-AGENT HTTP header.
- **appCodeName**
- **platform:** the hardware platform on which the browser is running.

### 2.3.2 Manipulating Windows

Youn can open a new web browser window with the **open()** method of the Window object. This is the method used to generate pop ups. So, most web browsers have now instuted some kind of pop-up-blocking system. **Window.open()** takes four optional arguments and returns a Window object that represents a newly opened window:

- The URL of the document to display in the new window.
- The name of the new window.

- A list of features that specify the window size and GUI decorations. If you omit this argument, the new window is given a default size and has a full set of standard features: a menu bar, a status line, a tool bar and so on. For security reasons, there are several restrictions on the features of the new window. For instance, it must not be too small, it must have a status line etc

- The fourth argument is used when the name that you provide for the new window is already in use. In this case, you must provide a boolean value to indicate whether the URL specified as the first argument should replace the current entry in the window's browsing history (true) or create a new entry in the windows's browsing history (false), which is the default behavior.

The result of the open method is a Window object that represents the newly created window. So, you have a reference to the newly created window and you can hold on to it. However, you can also pass to the new window a reference to the window in which it was created through the **opener** property.
You can open windows and you can close windows. If you hold a reference to a window you can just call the method **close()** on it. If you want to close the window in which the program is running you can call call **window.close()**. A window object still exists after being closed (the property **closed** is set to true). However, you cannot access its remaining properties. So, it is always a good idea to test if a window is closed (it may be closed by the user).
The Window object defines methods that move and resize a window. However, you should not use them: the user is the one who may resize the window at any time (not you!). Several methods:

- **movesTo()** moves the upper-left corner of the window to the specified coordinates

- **movesBy()** moves the window a specified number of pixels left or right and up or down.

- **resizeTo()** and **resizeBy()** resize the window by an absolute ore relative amount.

Note your ability to move and resize a window is restricted due to security constraints.
The **focus()** methods requests that the system gives keyboard focus to the window and the **blur()** method relinquishes it. Note that upon acquiring keyboard focus, the window is moved to the top of the stacking order.
There are several different techniques to scroll to a specific part of the page:

- **scrollsBy()** scrolls the document displayed on the window by a specific number of pixels left or right and up or down. **scrollsTo()** scrolls the document to an abolute position.

- When calling the function **focus()** on an element of the document, as part of transfering focus to the element, the document is scrolled to make the element visible. You can also use the methdod **scrollIntoView()** that is similar to the previous one, but additionally tries to put the element at the top of the window.
- You can place an **anchor** element in the place to which you want to transfer focus. For example, if you define an anchor named **top** at the start of the document, you can jump back to the top like this:

  ```
  window.location.hash = "#top"
  ```

  This technique leverages HTML's ability to travel within a document using named anchors. It makes the current document position visible in the browser's location bar, makes the location bookmarkable and allows the user to go back to his previous position with the **Back** button. If you don't want to clutter the browser's history just use the **replace()** method:

  ```
  window.location.replace("#top");
  ```

The window object provides three methods for displaying simple dialog boxes to the user:

- **alert()** displays a message to the user and ways for the user to dismiss the dialog.
- **confirm()** asks the user to click **OK** or **Cancel**.
- **prompt()** asks the user to enter a string.

These methods are not considered good web design practices. So do not use them. Moreover, the text displayed by these dialog boxes is plain text, not HTML formatted text. The **confirm()** and **prompt()** methods block. That is, they do not return until the user dismisses the dialog boxes they display.
Web browsers typically display a status line at the bottom of every window in which the browser can display messages to the user. When the user moves the mouse over a hypertext link, for example, a browser typically displays the URL to which the link points. In the past browsers allowed JavaScript code to set the

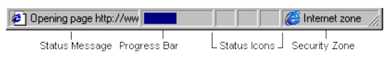

Figure 1: Status Line in IE

value of the property **status**. Current do not allow it, for security reasons (pg. 287). The Window object also defines a defaultStatus property that specifies thext to be displayed in the status line when the browser doesn't have anything else to display there.

If you assign a function to the **onerror** property of the Window object, the function is invoked whenever an error occurs in that window. Three arguments are passed to an error handler:

- A message describing the error.
- A string that contains the URL of the JavaScript document containing the error.
- The line number within the document where the error occurred.

If the **onerror** function returns true the browser will take no further action, otherwise the browser will display an error message.

### 2.3.3 Multiple Windows and Frames

It is possible to create web applications that use two or more frames or windows and use JavaScript code to make those windows or frames interact with one another. From a point of view of web design this is not a good approach and thus should be avoided.
Recall that the **open()** method of the Window object returns a reference to a new Window object. The new window object has an **opener** property that refers back to the original window. In this way, the two windows can refer to each other. Similarly, any frame in a window can refer to any other frame through the use of the **frames**, **parent** and **top** properties of the Window object. The JavaScript code in any window or frame can refer to its own window or frame as **window** or **self**. Every window and frame has a **frames** property that refers to an array of Window objects, each of which represents a frame contained within the window. Also, every window has a **parent** property, that refers to the Window object in which it is contained. The **top** property refers to the top-level containing window.
In HTML frames are created with the following tags: **frame**, **iframe** and **frameset**. When you create a frame with an HTML frame tag, you can specify a name for the frame with the **name** attribute. It is important to specify names for new frames because these names can be use in the **target** attribute of anchors and forms (the **target** attribute specifies where the results of following the link or submitting the form will be displayed). As far as JavaScript is concerned, if you specifie the name of the frame in the HTML file:

```
<frame name="table_of_contents" src="">
```

Then, the window objects which contains this frame will have the property **table_of_contents** that refers to the corresponding frame (note that the frame is also in the frames array).

```
parent.table_of_contents;
```

## 2.4 The Document Object Model

Consider the following example:

```
var today = new Date();
document.write("<p>A data de hoje e:" + d + "</p>");
```

You can use the **write()** method to output HTML to the current document only while that document is being parsed. You can call it from top-level code (code that is stored in the header) because this code is evaluated while the document is being parsed. If you call **write()** while handling an event you will erase the current document.
Legacy Dom properties:

- **bgColor:** the background color of the document.
- **cookie:** a special property that allows JavaScript programs to read and write HTTP cookies.
- **domain:** a property that allows mutually trusted web servers within the same Internet domain to collaboratively relax the same-origin policy security restrictions on interactions between their web-pages.
- **lastmodified:** a string that contains the modifications date of the document.
- **location:** a deprecated synonym for the **URL** property.
- **referrer:** the URL of the document containing the link, if any, that brought the browser to the current document.
- **title:** the text between title tags.
- **URL:** a string specifying the URL from which the document was loaded.

The array valued properties are the heart of the legacy DOM:

- **anchors[]:** an array of Anchor objects that represent the anchors in the document. The **name** property of an Anchor object holds the value of the name attribute.
- **applets[]:** an array of Applet objects that represent the applets in the document.
- **forms[]:** an array of From objects that represent the form elements in the document. Each form has its own array of elements: **elements[]**. Form objects trigger an **onsubmit** event handler before the form is submitted. This handler can perform client-side validation: if it returns **false** the browser will not submit the form.
- **images[]:** an array of Image objects that represent the images in the document. The **src** property of an Image object is read/write.

- **links[]:** an array of Link objects that represent the hypertext links in the document. The **href** property of the object corresponds to the **href** property of the anchor tag: the URL of the link. Link objects also make the various components of the URL available through properties such as **protocol**, **hostname** and **pathname**. The Link object triggers an **onmouseover** event handler and an **onmouseout** event handler. Additionally, it triggers an **onclick** event handler: if it returns false the browser will not follow the link.

The objects contained by these legacy DOM collections are scriptable, but it is important to understand that none of them allows you to alter the structure of the document because old browsers were not able to *reflow* (or *relayout*).
Elements are numbered with respect to the position in which they appear in the document. However, this is difficult way to refer to the document elements. Instead you can choose to name HTML elements using the name attribute and then refer to corresponding JavaScript objects by their name.

```
<form name="formularioMaravilha">
   <input name="inputinho" type="text" />
</form>

document.forms.formularioMaravilha
document.forms[0]
document.forms["formularioMaravilha"]

documents.forms.formularioMaravilha.inputinho
```

In fact, setting a name attribute form a form, image or applet (but not for an anchor element) also makes the corresponding form, image or applet accessible as a named property of the document object itself. Thus, we can write:

```
document.formularioMaravilha
```

However, if two elements have the same name (for instance, an image and a form), the document property corresponding to that name becomes an array with references to all objects sharing that name.
Every HTML element has a corresponding object in legacy DOM. Moreover, the attributes of each element also have corresponding JavaScript objects. For instance, you can set the **onsubmit** attribute of an HTML form to a string containing JavaScript code. Instead, you can set the **onclick** porperty of the JavaScript object corresponding to the form to the function that will handle the form.

### 2.4.1 W3C DOM

The tree representation of an HTML document contains nodes representing HTML tags, such as body and p, and nodes representing strings of text. So inside a **p** node there will probably be a **text** node containing the actual sentences

| **Interface** | **nodeType constant** | **nodeType value** |
|---|---|---|
| Element | Node.ELEMENT_NODE | 1 |
| Text | Node.TEXT_NODE | 2 |
| Document | Node.DOCUMENT_NODE | 3 |
| Comment | Node.COMMENT_NODE | 8 |
| DocummentFragment | Node.DOCUMMENT_FRAGMENT_NODE | 11 |
| Attr | Node.ATTRIBUTE_NODE | 2 |

of the paragraph. Recall tree terminology: parent, children, sibling, descendants and ancestors.
The DOM **Node** interface defines properties and methods for traversing and manipulating the tree. The **childNodes** property of a Node object returns a list of children of the node, and the **firstChild**, **lastChild**, **nextSibling**, **previousSibling** and **parentNode** provide a way to traverse the tree of nodes. Methods such as **appendChild**, **removeChild**, **replaceChild** and **insertBefor()** enable you to add or remove nodes from the document tree.
Different types of nodes in the document tree are represented by specific subinterfaces of Node. Every Node object has a **nodeTyep** property that specifies what kind of node it is. Types of nodes:

The Node at the root of the DOM tree is a Document object. The **documentElement** property of this object refers to an Element object that represents the root element of the document. In HTML documents, to access the body element directly one can write: **document.body**. There is only one Document object in a DOM tree. Most nodes in the tree are Element objects that represent tags which appear within the document and Text objects which represent strings of text. If the document parser preserves comments this comments are represented in the DOM tree by Comment objects. The attributes of an element may be queried set and deleted using the methods of the Element interface: **getAttribute()**, **setAttribute()** and **removeAttribute()**. It is important to note that attributes also correspond to nodes: one can use the method **getAttributeNode()** that returns an Attr object representing the node and its value. The DOM specification allows the attributes to be access through the **attributes**[] array of the Node interface.
The DOM was designed for use with XML and HTML. So the DOM API is relatively generic. Nevertheless, the DOM standard includes interfaces that are specific to HTML documents. The HTMLDocument interface defines various document properties and methods that were supported by browsers prior to W3C standardization. These include the **location** property, **forms**[] etc. The HTMLElement interface defines **id**, **style**, **title**, **lang**, **dir**, **className** properties. These properties allow convenient access to the values of the corresponding HTML attributes. All other HTML tags have corresponding interfaces defined by the HTML portion of the DOM specification. Note that the DOM standard defines properties for HTML attributes as a convenience for script writers. The general way to query and set attribute values is with the **getAttribute()** and

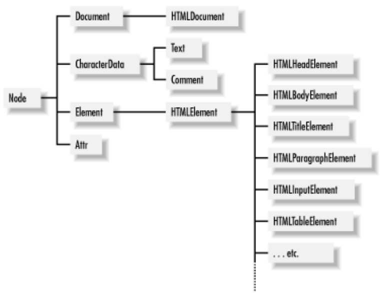

Figure 2: DOM API

s

**setAttribute()** methods of the Element object.
Not all browsers support the complete W3C specification (of course). One source of comformance information is the implementation itself. In a conformant implementation, the **implementation** property of the Document object refers to a DOMImplementation object that defines a method named **hasFeature**.

```
if(document.implementation &&
   document.implementation.hasFeature &&
   document.implementation.hasFeature("html", "1.0")) {
   //The browser claims to support level 1 Core and Html intefaces
}
```

Check page 314 of the book.
It is important to understand that the DOM API is exactly what it claims to be - an API. Each browser implements the API in its own way. Furthermore, the interfaces Element, Attribute, Document etc are exactly what they claim to be - interfaces. So, you cannot be sure there is a constructor element for each one of these interfaces. Since it cannot define constructors the DOM standard instead defines a number of useful *factory methods* in the Document Interface. So, to create a new Text object for a document:

```
var t = document.createTextNode("this is a new text node");
```

Factory methods defined by the DOM have names that begin with the word 'create'.
Traversing a document:

```
  function countTags(n) {
    var numtags = 0;
    if(n.nodeType == NODE.ELEMENT_NODE) numtags++;
var children = n.childNodes;
for(var i=0; i<children.length; i++)
  numTags += countTags(children[i]);
return numTags;
  }
```

You cannot traverse or manipulate de document tree until the document has been fully loaded. Another example:

```
function getText(n) {
  var strings = [];
  getStrings(n, strings);
  return strings.join("");
}

function getStrings(n, strings) {
  if(n.nodeType == Node.TEXT_NODE) strings.push(n.data);
  if(n.nodeType == Node.ELEMENT_NODE)
   for(var m = n.firstChild; m!= null; m)
     getStrings(m, strings);
}
```

The Document object is the root of every DOM tree, but it does not represent an HTML element in that tree (it represents the document itself). The **document.documentElement** property refers to the html tag that serves as the root element of the document. And the **document.body** property refers to the body tag.
In order to find specific elements within an document, the HTMLDocument interface provides the following methods: **getElementById()**, **getElementByTagName()** and **getElementByName()**. The DOM Element interface only provides the first two methods. You may also want to get all the elements from a given class. There is no method in the DOM Element interface to achieve this goal, however you can check the **classname** property of an element (check the implementation in page 323). Warning: an element may belong to several classes.

```
<p class="aviso informacoes">....</p>
```

Traversing a document:

- **childNodes:** a property of every Node.
- **firstChild**
- **lastChild**
- **nextSibling**

Modifying a document:

- **appendChild()** is part of the DOM node interface and adds the node passed as an argument to the children of the node on which it is called. When we insert a node that is already part of the document, it is automatically removed from its current postion.

- **replaceChild()**.
- **createElement()**. This method is provided by the Document object.
- **insertBefore()**.

Manipulating text nodes:

- Manipulate the **data** property itself.
- **appendData()**
- **insertData()**
- **deleteData()**
- **replaceData()**

Creating nodes:

- **Document.createElement()**
- **Document.createTextNode()**

Modifying attributes. Each element provides the method **setAttribute()** in order to change the value of one of its attributes. The DOM elements that represent HTML elements define JavaScript properties for the corresponding HTML attributes, so:

```
var headline = document.getElementById("headline");
headline.setAttribute("align", "center");

var headline = document.getElementById("headline");
headline.align = "center";
```

A DocumentFragment is a special type of node that does not appear in a document itself but serves as a temporary for a sequential collection of nodes and allows those nodes to be manipulated as a single object. When a DocumentFragment is inserted into a document (using the replaceChild(), appendChild(), insertBefore() methods of the Node object), it is not the DocumentFragment itself that is inserted, but each of its children. You can create a DocumentFragment with **document.createDocumentFragment()**.

```
function reverse(n) {
  var f = document.createDocumentFragment();
  while(n.lastChild) f.appendChild(n.lastChild);
  n.appendChild(f);
}
```

Systematizing the creation of an element:

1. Create the element invoking **document.createElement()**. This method receives as an argument the type of element you want to create.
2. Then, you must set its attributes appropriately. As you know, HTML element attributes correspond to DOM HTMLElement object properties.
3. Then, you create a Text node using **document.createTextNode()**.
4. Next, you had the element to its parent element invoking **appendChild()**.

The **innerHTML** property is not officially part of the DOM, however it is implemented by every browser. When you query the value of this property for an HTML element, you get a string of HTML text that represents the children of this element. You can also explicitly set this property. This may be an alternatie to the procedure described above.

## 2.5 Scripting CSS and Dynamic HTML

For DHTML developers, the most important feature of CSS is the ability to use ordinary CSS style attributes to specify the visibility, size and precise position of individual elements of a document.

- **position:** specifies the type of the positioning applied to an element.
- **top, left:** specify the position of the top and left edges of a document.
- **bottom, right:** specify the position of the bottom and right edges of an element.
- **width, height:** specify the size of an element.
- **z-index:** specifies the stacking order of an element to any overlapping elements.
- **display:** specifies how and whether an element is displayed.
- **visibility:** specifies whether an element is visible.
- **clip:** defines a clipping region for an element; only portions of the element within this region are displayed.
- **overflow:** specifies what to do if the element is bigger than the space alloted for it.
- **margin, border, padding:** specify spacing and borders for an element.
- **background:** specifies the background color or image for an element.
- **opacity:** specifies how opaque an element is.

The CSS **position** attribute specifies the type of positioning applied to an element:

- **static:** This is the default value and specifies that the element is positioned according to the normal flow of the document content. Statically postioned elements are not DHTML elements and cannot be positioned with **top**, **left** and other attributes.
- **absolute:** This value allows you to specify the position of an element relative to its containing element. Absolutely positioned elements are positioned independently of all other elements and are not part of the flow of statically positioned elements.
- **fixed:** This value allows you to specify the postion of an element with respect to the browser window. Elements with **fixed** positioning are always fixed and do not scroll with the rest of the document.
- **relative:** When the **position** attribute is set to **relative**, an element is laid out according to the normal flow and its position is then adjusted relative to its position in the normal flow. The space allocated for the element in the normal document flow remains allocated for it, and the elements on either side of it do not close up to fill in the space, nor are they pushed away from the new position of the element.

Once you have set the **position** attribute to a value other than **static**, you can set the new position with a combination of the **left, top, right, bottom** attributes.

```
<div style="position: absolute; left: 100px; top: 100px;">
```

The containing element relative to which a dynamic element is positioned is not necessarily the same as the containing element within which the element is defined in the document source. Since dynamic elements are not part of normal event flow, their positions are not specified with respect to the static container element within which they are defined. Most dynamic elements are positioned relative to the document (the body tag) itself. The exception is dynamic elements that are defined within other dynamic elements. In this case the nested dynamic element is positioned relative to its nearest dynamic ancestor. If you whish to position an element relative to a container that is part of the normal document flow, set its **position** attribute to **relative** and then specify a **top** and **left** position of 0px.
In addition to the position of elements, CSS allows you specify their size. This is most commonly done by providing values for the **width** and **height** style attributes.

```
<div style="position: absolute; top: 10px; left:10px;
             width: 10px; height: 10px; background-color: blue">
</div>
```

If you specify the **right**, **left** and **width**, the **width** overrides the **right**. If you specify the **top**, **bottom** and **heigth** attributes, the **height** overrides the **bottom**.
The standard CSS allows measures to be done in a number of different units:

- **px:** pixels
- **in:** inches
- **pt:** points
- **cm:** centimeters
- **em:** a measure of the line height for the current font

Instead of specifying absolute postions and sizes using the units shown above, CSS allows you to specify the position and size of an element as a percentage of the size of the containing element.

```
<div style="position: absolute; left: 25%; top: 25%; width: 50%; height: 50%;
            border: 2px solid black">
```

To script CSS styles you can use the method: **setAttribute()** in order to set the attribute **style** of the HTML element to which you want to apply the new style.

```
element.style.position = "relative";

var shadow = document.createElement("span");
shadow.setAttribute("style", "position: absolute; " +
                             "left: " + "shadowX + ";" +
                             "top: " + "shadowY + ";" +
                             "color: " + "shadowColor + ";");
```

In moder browsers, the **offsetLeft** and **offsetTop** properties of an element return the X and Y coordinates of the element relative to some other element. This element is the value of the **offsetParent** property. For positioned elements, the **offsetParent** is typically the body or a positioned ancestor of the **offsetParent**. For nonpositioned elements, different browsers handle the **offsetParent** differently. Clearly, the **offsetLeft** and **offsetTop** elements mirror the CSS attributes **Left** and **top**.

```
function getX(e) {
  var x = 0;
  while(e != null) {
     x += e.offsetLeft;
     e = e.offsetParent;
  }
}
```

The **offsetWidth** and **offsetHeight** properties return the width and height. These properties are read-only and return pixel values as numbers.
Two CSS atributes affect the visibility of a document element: **visibility** and **display**. When the **visibility** attribute is set to the value **hidden**, the element is not shown. However, if the element appears in the normal flow layout the

element only becomes invisible its space is not occupied by the remaining elements. The **display** attribute is used to specify the type of display an item receives. When the **display** element is set to **none**, the affected element is not displayed or even layed out. You will typically use the **visibility** attribute when you are working with dynamically positioned elements. The **display** attribute is usefull when creating expanding and collapsing outlines.
The **z-index** attribute is an integer. The default value is zero, but you may specify positive or negative values. When two or more elements overlap, they are drawn in order from lowest to highest z-index; the element with the highest z-index appears on top of all others. If overlapping elements have the same **z-index**, they are drawn in the order they appear in the document (so the last overlapping element appears on top). Note tha z-index stacking only applies to sibling elements. If two elements that are not siblings overlap, the browser decides which element to show according to the z-index of their containers. Nonpositioned elements are always laid out in a way that prevents overlaps. So, the z-index does not apply to them. Nevertheless, they have a default z-index of zero, which means that positioned elements with a positive z-index appear on top of the normal document flow and positioned elements with a negative z-index appear below the normal document flow.

### 2.5.1 The CSS Box Model

CSS allows you to specify margins, borders and padding for any element and this complicates CSS positioning because you have to know how **width**, **heigth**, **top** and **left** are calculated in the presence of these components.

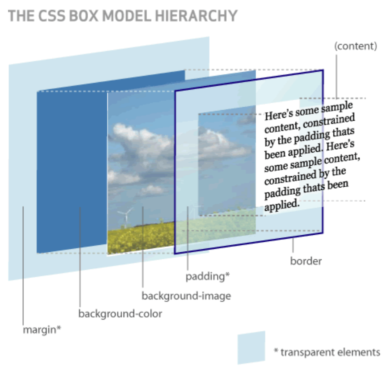
THE CSS BOX MODEL HIERARCHY
(content)
Here's some sample content, constrained by the padding thats been applied. Here's some sample content, constrained by the padding thats been applied.
border
padding*
background-image
background-color
margin*
* transparent elements

Figure 3: The Box Model

The **border** of an element is a rectangle *drawn* around it. CSS attributes allow you to specify the style, color, and thickness of the border:

```
border: solid black 1px;
border: 3px dotted red;
```

It is possible to specify the border width, style and color using individual CSS attributes (border-width, border-color and border-style). Addtionally, you can specify the border attributes for individual sides of an element. Each element has four sides: top, bottom, left and right. So, to customize the top border you can use border-top, border-top-color, border-top-style, border-top-width. The **margin** and **padding** attributes both specify blank space around an element. The difference is that the **marign** specifies space outside the border, between the border and the adjacent elements, and **padding** specifies space inside the border and the content area. A margin provides visual space between an element and its neighbors in the normal document flow. Padding keeps element content visually separated from its border. If an element has no border padding is not necessary. If an element is dynamically positioned, it is not part of the normal document flow, and its margins are irrelevant. You can specify the margins and padding of an element with the attributes **margin** and **padding**.

```
margin: 5px; padding: 5px;
```

You can also specify margins and paddings for individual sides of an element:

```
margin-top: 5px; padding-bottom: 10px;
```

You can also specify the margin and padding values for all four edges simultaneously (TROUBLE mnemonic):

```
margin: 5px 3px 5px 2px;
padding: 3px 5px 1px 8px;
```

It is important to understand that the **width** and **height** attributes refer to the width and the height of the content area. Thus, you might think that the **left** and **top** (and **right** and **bottom**) would be measured relative to the content area of the containing element. This is not the case, though. The CSS specifies that these values are measured relative to the outside edge of the containing element's padding (from the margin of the contained element to the inside of the border of the containing element).
The **visibility** attribute allows you to completely hide a document element. The **overflow** and **clip** attributes allow you to display only part of an element. The **overflow** element specifies what happens when the content of an element exceeds the size specified:

- visible: Content may overflow and be drawn outside of the element's box if necessary.
- hidden: Content that overflows is clipped and hidden.
- scroll: The elements box has a permanent horizontal and vertical scrollbar.
- auto: Scrollbars are displayed only when content exceeds the element's size.

The **clip** property specifies the clipping region for the element. The syntax of the clip is the following: rect(top right bottom left). The top, right, bottom, left values specify the boundaries of the clipping rectangle relative to the upper-left corner of the element's box. You may use the auto keyword for any of the four values to specify that the edge of the clipping region is the same as the corresponding edge of the element's box.

```
clip: rect(auto 100px auto auto);
clip: rect(0px 100px 100px 100px 0px);
```

### 2.5.2 Color

You can specify colors using english names of common colors. You can also specify colors hexadecimal notation (check a table).

- **color:** color of the foreground elements (the foreground of an element consists of its text and border).
- **background-color:** color of the background.
- **border-color:** color of the border.

### 2.5.3 Background

You can specify an image as the background of an element with the attribute: **background-image**. The attributes **background-attachment**, **background-position**, **background-repeat** attributes specify further details about the image is drawn. The shortcut **background** allows you to specify these attributes together. Like background colors, tiling background images fill the area behind the content, the padding and extend to the outer edge of the border.

```
body { background-image: url(star.gif); }
```

Images tile up and down, left and right. However, this behavior can be changed setting the **background-repeat** property: repeat, repeat-x, repeat-y, no-repeat, inherit. The **background-position** property specifies the position of the image in the background:

- **keyword positioning:** The keyword values (left, center, right, top, bottom and center) position the image relative to the edges of the element.

  ```
  { background-position: left bottom; }
  ```

- **Length measurements:** you can specify the postion by its distance from the top-level corner of the element using pixel measurements. When providing length values the horizontal element always goes first.

  ```
  { background-position: 200px 50px; }
  ```

- You can also use percentages.

```
{ background-position: 15% 30%; }
```

With the **background-attachment** property you have the choice whether the background image scrolls (**scroll**) or is fixed (**fixed**).
You can the **background** property to specify all your background styles in one declaration.

```
body { background: black url(arlo.jpg) no-repeat right top fixed; }
```

It is important to uderstand that if you do not specify a background color or image for an element, that element's background is usually transparent. You can also specify translucent elements using the **opacity** CSS property:

```
opacity: 0.75;
```

### 2.5.4 Scripting Inline styles

The crux of DHTML is its ability to use JavaScript to dynamically change the style attributes applied to individual elements within a document. To manipulate the style of an HTML element, you only have to obtain a reference to it, and then you can use the element's **style** property. It is important to understand that the CSS2Properties object you obtain with the **style** property of an element specifies only the inle styles of the element. So, the values of the properties of a CSS2Properties object are only meaningfull if they were previously set by your JavaScript code. You cannot use the CSS2Properties object to obtain information about the stylesheet styles that apply to the element. By setting properties on this object, you are defining inline styles taht effectively override stylesheet styles.

```
var imgs = document.getElementsByTagName("img");
for(var i=0; i<imgs.length; i++){
  var img = imgs[i];
  if(img.width == 468 && img.heigh == 60)
    img.style.visibility = "hidden";
}
```

The following example illustrates a very simple animation.

```
<script>
var e = document.getElementById("urgent");
e.style.border = "solid black 5px";
e.style.padding = "50px";
var colors = ["white", "yellow", "orange", "red"];
var nextColor = 0;
setInterval(function() {
          e.style.borderColor = colors[nextColor++%color.lenght];
        }, 500);
</script>
```

The W3C standard API for determining the *computed style* of an element is the **getComputedStyle()** method of the Window object. This method returns a read-only object. The IE does not support the **getComputedStyle()** method. Instead, every HTML element has a **currentStyle** property that holds its computed style.

```
var p = document.getElementsByTagName("p").[0];
var typeface = "";
if(p.currentStyle)
  typeface = p.currentStyle.fontFamily;
else if(window.getComputedStyle)
  typeface = window.getComputedStyle(p, null).fontFamily;
```

Each HTMLElement object has a **className** property that mirror the HTML **class** attribute. Of course, you can set the value of this property and change the presentation of the element but be careful.

```
e.className = "classeMagica";
```

The HTML DOM Level 2 defines a **disabled** property for both **link** and **style** elements. There is no corresponding **disabled** attribute on HTML tags, but there is a property that you can query and set in JavaScript. As its name implies, if the disabled property is true, the stylesheet related to the link or style element is disabled and ignored by the browser.

```
enableSS('ss1', this.checked);
```

To fully understand this example check page: 383.
The HTML DOM Level 2 also defines a complete API for querying, traversing and manipulating stylesheets. The stylesheets that apply to a document are stored in the **stylesheets**[] array of the document object. A **CSSStylesheet** object has a **cssRules**[] array that contains the rules of the stylesheet:

```
var firstRule = document.styleSheets[0].cssRules[0];
```

The elements of the **cssRules**[] array are **CSSRule** objects. Each of these objects comprises two important properties: **selectorText** is the CSS selector for the rule and the **style** refers to a CSS2Properties object. Recall that **CSS2Properties** is the same interface that is used to represent inline styles of an HTML element through the **style** property. You can use this **CSS2Properties** object to query the style values or to set new styles for the rule. The **cssText** property of a **CSSRule** object refers to the string representation of the rule. Naturally, you can add new rules to a **CSSStyleSheet** object using the method **insertRule()** and remover a rule using the method **deleteRule()**.

```
document.styleSheets[0].insertRule("H1 { text-weight: bold; }", 0);
```

## 2.6 Events and Event Handling

Interactive JavaScript programs use an *event* driven programming model. In this style of programming, the web browser generates an event whenever something interesting happens to the document or to some element of it. If a JavaScript application cares about a particular type of event for a particular document element, it can register an *event handler* - a JavaScript function or snippet of code for that type of event on the element of interest. Then, when the particular event occurs the browser invokes the handler code.
The envent handling models in use are the following:

- *The original event model* is considered part of the DOM level 0 API and is supported by all JavaScript-enabled web browsers.
- *The standard event model* is a powerfull and full-featured event model that was standardized by the DOM Level 2 API. It is supported by all moder browsers except Internet Explorer.
- *The Internet Explorer Event Model* was introduced in IE 4 and was extended in IE 5.

### 2.6.1 Basic Event Handling

In the orginal *event model*, an event is an abstraction internal to the web browser and JavaScript code cannot manipulate an event directly. When we speak of an *event type* in the original event model what we really mean is the name of the event handler that is invoked in response to the event. Table **??**ists all the event-handler attributes that you can use in the original event model. It also specifies when these events are triggered and which HTML elements spport the handler attributes. Mouse event handlers are supported by most HTML elements (the elements that do not support this event are typically elements that belong in the **head** of a document, or do not have a graphical representation of their own.

There are two different categories of events: *raw events* and *semantic events*. *Raw events* or *input events* are the events that are generated when the user moves or clicks the mouse or presses a key on the keyboard. These low-level events simply describe a user's gesture and have no other meaning. *Semantica events* are higher-level events that have a more complex meaning and can typically occur only in specific contexts: when a browser has finished loading a document or when a form is about ot be submitted. A semantic event often occurs as a side-effect of a lower-level event. For example, when the user clicks the mouse over the **submit** button, three of the button's input handler are triggered: **onmousedown**, **onmouseup** and **onclick**. Then, as a result of this mouse click, the HTML form that contains the **Submit** button generates an **onsubmit** event.

Another important distinctions divides events into *device-dependent* events and *device-independent* events. This distinction is particularly important for accessibility purposes because some users may be able to use a mouse but not a keyboard etc. Semantic events, such as **onsubmit** and **onload**, are almost

| | |
|---|---|
| onabort | Image loading interrupted. |
| onblur | Element loses input focus. |
| onchange | Selection in a select element or other form element loses focus and its value has changed sin |
| onclick | Mouse press and realese. |
| ondbclick | Double-click. |
| onerror | Error loading image. |
| onfocus | Element gains input focus. |
| onkeydown | Key pressed down. |
| onkeypressed | Key pressed; follows key down. |
| onkeyup | Key realesed; follows key pressed. |
| onload | Document load complete. |
| onmousedown | Mouse button pressed. |
| onmousemove | Mouse moved. |
| onmouseout | Mouse moves off element. |
| onmouseover | Mouse moves over element. |
| onmouseup | Mouse button realesed. |
| onreset | Form reset requested. Return false to prevent reset. |
| onresize | Window size changes. |
| onselect | Text selected. |
| onsubmit | Form submission requested. Return false to prevent submission |
| onunload | Document or frameset unloaded. |

always device-independent events. Note that all moder browsers allow the user to manipulate forms using the mouse or keyboard traversal. So, if you choose to use device-dependent events, events that have the word mouse or key, use them in pairs so that you provide a handler for a mouse gesture and a keyboard alternative. The **onclick** event can be considered a device-independent event since keybord activation of form controls and hyperlinks also generates this event.

Events handlers are specified as strings of JavaScript code used for the values of HTML attributes. So, for example, to execute JavaScript code when the user clicks a button, specify that code as the value of the **onclick** attribute of the **input** tag:

```
<input type="button" value="Click Me"
      onClick = "alert(cabrao);" />
```

Obviously, when an event handler requeires multiple statements, it is usually easier to define them in the body of a function and then use the HTML event handler attribute to invoke that function. Another example, this time concerning form validation:

```
<form action="'processform.cgi" onsubmit="return validateForm();">
```

Remember that HTML is case-insensitive so you can capitalize event handler attributes any way you choose.

Each HTML eleemnt in a document has a corresponding DOM element in the document tree, and the properties of this JavaScript object correspond to the attributes of the HTML element. This applies to event-handler attributes as well. So, if an **input** tag has an **onclick** attribute, the event handler it contains can be referred to with the **onclick** property of the corresponding DOM object. When accessed through JavaScript event-handler properties are functions. Therefore, to assign an event-handler to a document element using JavaScript, simply set the event-handler property to the desired function.

```
HTML:
<form name="f1">
  <input name="b1" type="button" value="Press Me" />
</form>

JavaScript:
document.f1.b1.onclick = function(){alert('Thanks'); };
```

Advantages of defining event handlers as JavaScript properties:

- Reduces intermingling between JavaScript and HTML, promoting modularity and cleaner, more maintainable code.
- Allows event handler functions to be dynamic.
- It can sometimes be usefull to change the event handlers registered for HTML elements.

Disadvantages of defining event handlers as JavaScript properties:

- It separates the handler from the element from which it belongs. If the user interacts with a document element before the document is fully loaded (and thus before all its scripts have been executed), the event handlers for the document element may not be defined yet.

```
function confirmLink() {
  return confirm("Do you really want to visit " + this.href + "?");
}

function confirmAllLinks(){
  for (var i=0;  i<document.links.length; i++){
     document.links[i].onclick = confirmLink;
  }
}
```

We note that if the function **confirmLink** returns false, the browser will not follow the link.

Because the values of JavaScript event handler properties are functions, you can use JavaScript to invoke event handler functions directly.

```
  document.myform.onsubmit();
```

Note however, that invoking an event handler directly is not a way to simulate what happens when the actual event occurs. If you invoke the onclick method of a Link object for example it does not make the browser follow the link and load a new document. However, if you call the **submit()** method of a form object, you will actually submit the Form. Another example:

```
  var b = document.myform.mybutton;
  var oldHandler = b.onclick;
  var newHandler = function(){ ... };
  b.onclick = function(){ oldHandler(); newHandler(); }
```

You can use the return of an event handler to indicate the disposition of the event, that is, if the web browser performs some kind of default action in response to an event, you can return false to prevent the browser from performing that action. Example:

```
 <form action="search.cgi" o
       onsubmit="if(this.elements[0].value.lenght == 0) return false;">
 </form>
```

When an event handler is invoked, it is invoked as a method of the element on which the event occurred, so the this keyword refers to that target element. Nothing special here!!!!

The next object in the *scope chain* of event handlers defined as HTML attributes is the object that triggered the event. Check the following example:

```
<form>
<!-- the this keyword refers to the input element -->
<input id="b1" type="button" value="button 1"
       onclick = "alert(this.form.b2.value);"/>
<!-- the next example works because the input element is in the scope chain -->
<input id="b2" type="button" value="button 2"
       onclick = "alert(form.b1.value);" />
<!-- the next example works because the form object is in the scope chain -->
<input id="b3" type="button" value="button 3"
       onclick = "alert(b1.value);" />
<!-- the document object is in the scope chain, so we can use its methods without
     prefixing them with document -->
<input id="b4" type="button" value="button 4"
       onclick = "alert(getElementById('b3').value);"/>
</form>
```

The moral is that you must be careful when defining event handlers as HTML attributes. Your safest bet is to keep any such handlers very simple. Ideally, they should just call a global function defined elsewhere and perhaps return the result.

```
<script> function validateForm(){/* code here*/} </script>
<form onsubmit = "return validateForm();">...</form>
```

### 2.6.2 The DOM Level 2 Event Model

In the Level 0 event model, the browser dispatches elements to the document elements on which they occur. If that object has an appropriate event handler, that handler is run. In the DOM Level 2 event model, event propagation proceeds in three phases. First, during the *capturing phase*, events propagate from the Document object down through the document tree to the target node. If any ancestors of the target (but not the target itself) has a specially registered capturing event handler, those hanlders are run during this phase of event propagation. The next phase of event propagation occurs at the target node itself: any appropriate event handlers registered directly on the target are run. The third phase of event propagation is the *bubbling phase*, in which the event propagates or *bubbles* back up the document hierarchy from the target element up to the Document object. Although all events are subject to the capturing phase, not all types of events bubble.

During the event propagation, it is possible for any event handler to stop further propagation of the event by calling the **stopPropagation()** method of the Event object.

Some events cause an associated default action to be performed by the web browser. For example, when a click event occurs on an anchor tag, the browser's default action is to follow the hyperlink. Default actions like these are performed only after all three phases of event propagation complete, and any of the handlers invoked during event propagation can prevent the default action from occurring by calling the **preventDefault()** method of the Event object.

In DOM Level 0 event model you can only register a single event handler for a particular type of event for a particular object. In DOM Level 2 event model you can register any number of handler functions for a particular event type on a particular object.

On of the advantages of the DOM Level 2 event model is that it allows you to produce simpler code. Suppose you want to trigger an event handler each time the user moves a mouse over a paragraph. Instead of regestering an onmouseover event for each paragraph, you can instead register a single event handler on the Document object and handle these events during the capturing or bubbling phase of event propagation.

To register an event handler for a particular element you need to call the method **addEventListener()** of that object. This method takes three arguments:

- The name of the event type for which the handler is being registered. The event type should be a string that contains the lowercase name of the HTML handler attribute, wht the leading "on" removed.
- The function to be invoked when the specified type of event occurs. When your function is invoked, it is passed an Event object as its only argument.
- A boolean that specifies if the handler captures events during the capturing phase of event propagation (nothing is said however about the bubbling phase).

```
document.myform.addEventListener("submit",
                                 function(e) {return validate(e.target);},
                                 false);

var mydiv = document.getElementById("mydiv");
mydiv.addEventListener("mousedown", handleMouseDown, true);
```

Note that event handlers registered with **addEventListener()** are executed in the scope they are defined. They are not invoked with the augmented scope chain.

As was already said, you can register more than one event handler for a particular type of event on a single object. Why would you want that? For modularity reasons...

The method **addEventListener()** is paired with the method **removeEventListener()**. It expects the same three arguments but removes an event-handler function from an object rather than adding it. It is useful to register an event handler and then remove it soon afterward.

```
document.removeEventListener("mousemove", handleMouseMove, true);
document.removeEventListener("mouseup", handleMouseUp, true);
```

In the DOM Level 0 event model, when a function is registered as an event handler for a document element, ti becomes a method of that document element. When the event handler is invoked it is invoked as method of that element, and, within the function, the this keyword refers to the element on which the event occurred. The DOM Level 2 event model says that event listeners should be objects rather than functions. In practice they are functions. But they are bound to which object? This is not specified. Generally, the this keyword refers to the object on which the handler was registered. If you do not want to rely on this unspecified behavior you can use the **current target** property of the Event object that is passed to your handler functions.

You can register an object as event handler rather than a function. This object must implement the **EventListener** interface and have a method named **handleEvent()**. This is all bullshit. You cannot pass directly an object to the function addEventListener(). You cannot. This is not Java programming. pg. 403.

When an event occurs, its handler is passed an object that implements the event. The properties of this object provide details that may be useful to the handler. There are three distinct Event interfaces: **Event**, **MouseEvent** and **UIEvents**.

- Events: abort, blur, change, error, focus, load, reset, resize, scroll, select, submit unload.
- MouseEvents: click, mousedown, mousemove, mouseout, mouseover, mouseup
- UIEvent: DOMActivate, DOMFocusIn, DOMFocusOut

Note that there does not exist an interface for keyboard events. DOM Level 2 does not standardize any type of keyboard event. Additionally, observe that UIEvent is a subinterface of Event and MouseEvent is a subinterface of UIEvent.

The Event interface:

- **type:** The type of event that occurred. The value of this property is a string and is the same string value that was used when regestering the event.
- **target:** The node on which the event occurred, which may not be the same as the **currentTarget**.
- **currentTarget:** The node at which the event is currently being processed.
- **eventPhase:** A number that specifies what pahse of event propagation is currently in process. The value is one of the constants: Event.CAPTURINGPAHSE, Event.ATTARGET, or Event.BUBBLINGPHASE.
- **timeStamp:** A Date object that specifies when the event occurred.
- **bubbles:** A boolean that specifies whether the event bubbles up the document tree.
- **cancelable:** A boolean that specifies whether the event has a *default action* associated with it that can be canceled with the **preventDefault()** method.

In additon to these properties, teh Event interface defines two methods that are also implemented by all event objects: **stopPropagation()** and **preventDefault()**.

The UIEvent interface:

- **view:** The Window object within which the event occurred.
- **detail:** A number that provides additional information about the event.

The MouseEvent interface:

- **button:** A number that specifies which mouse button changed during a mousedown, mouseup, or click event.
- **altKey, ctrlKey, metaKey, shiftKey:** These four boolean fields indicate whether the Alt, Ctrl, Meta, or Shift keys were held when a mouse event occurred.
- **clientX, clientY:** X and Y coordinates of the mouse pointer relative to the browser window.
- **screenX, screenY:** These two properties specify the X and Y coordinates of the mouse pointer relative to the upper-left corner of the user's monitor.

- **relatedTarget:**

Note that browsers that support the DOM Level 2 event model also support the DOM Level 0 event model. However, when an event handler is defined as an HTML attribute, it is implicitly converted to a function that has an argument named event. This means that such an event handler can use the identifier event to refer to that event object.

## 2.7 Form Processing

In server-side programming, when dealing with form processing the emphasis is on processing a complete batch of input data and dynamically producing a new web page in response. With JavaScript, the emphasis is not on form submission and processing but instead on event handling.

The JavaScript Form object represents an HTML form. Form objects are available as elements of the **forms**[] array, which is a property of the Document object. Forms appear in this array in the order in which they appear within the Document. The most interesting porperty of the Form object is the **elements**[] array, which contains JavaScript objects that represent the various input elements of the form. Again, this elements appear in the array in the order in which they appear in the form. The **action**, **encoding**, **method** and **target** properties of the Form object correspond to the action, encoding, method and target attributes of the Form tag.

In the days before JavaScript, a form was submitted with a special-purpose **Submit** button, and form elements had their values reset with a special-purpose **Reset** button. The JavaScript Form object supports two methods, **submit()** and **reset()**, that server the same purpose. Invoking the submit() method of a Form submits the form, and invoking the reset() method resets the form elements.

To accompany the submit() and reset() methods, the Form object provides the **onsubmit** event handler to detect form submission and the **onreset** event handler to detect form resets. The onsubmit event handler is invoked just before the form is submitted and it can cancel the submission by returning false. This provides an opportunity for a JavaScript program to check the user's input for errors in order to avoid submitting incomplete or invalid data over the network to a server-side program. Notice that the onsubmit handler is triggered only by a genuine click on a Submit button. Calling the submit() method of a form does not trigger the onsubmit handler. The onreset event handler is similar to the onsubmit handler. It is invoked just before the form is reset, and it can prevent the form elements from being reset by returning false. This allows a JavaScript program to ask for confirmation of the reset, which can be a good idea when the form is long or detailed. Like the onsubmit handler, the onreset handler is only invoked when the Reset button is pressed. Calling the reset() method does not trigger the onreset event.

```
<form ... onreset="return confirm('Do you really want to reset?');">
```

Consider the following example:

```
<html>
<head>
  <title>Playing with forms</title>
  <style type="text/css">
      td {border: 2px solid; }

  </style>

  <script type="text/javascript">
     function report(element, event){
   var value;
   if((element.type == 'select-one') || (element.type == 'select-multiple')){
      value = '';
          for(var i = 0; i<element.options.length; i++)
     if(element.options[i].selected)
    value += element.options[i].value + ' ';
   }
   else if(element.type == 'textarea') value = '...';
   else value = element.value;
   var msg = event + ': ' + element.name + ' (' + value + ')\n';
   var t = element.form.events;
   t.value = t.value + msg;
 }

 function addhandlers(f) {
    for(var i=0; i < f.elements.length; i++){
   var e = f.elements[i];
   e.onclick = function() { report(this, 'Click'); }
           e.onchange = function() { report(this, 'Change'); }
           e.onfocus = function() { report(this, 'Focus'); }
           e.onblur = function () {report(this, 'Blur'); }
           e.onselect = function() {report(this, 'Select'); }
}

f.clearbutton.onclick = function(){
                           this.form.events.value = "";
   report(this, 'Click');
}

f.submitbutton.onclick = function() {
                            report(this, 'Click');
return false;
     }

f.resetbutton.onclick = function() {
                            if(confirm('Do you really want to reset?'))
   this.form.reset();
report(this, 'Click');
return false;
                        }
 }

  </script>
</head>
<body>
 <h1>Playing with forms - and doing uncool stuff</h1>
 <form name="magicalform">
   <table id="tableMagicalForm">
     <tr>
     <td>Username:<br />
     [1]<input type="text" name="username" size="15" />
 </td>
 <td>Password:<br />
     [2]<input type="password" name="password" size="15" />
     </td>
 <td rowspan="4">Input Events[3]: <br />
     <textarea name="events" cols="28" rows="20">Escreva aqui os seus eventos!!</textarea>
     </td>
 <td rowspan="4">
   <fieldset>
    <legend>Important! </legend>
    [4]<input type="button" name="clearButton" value="Clear" /> <br />
[5]<input type="submit" name="submitButton" value="Submit"/> <br />
[6]<input type="reset" name="resetButton" value="Reset"/>
   </fieldset>
 </td>
     </tr>
     <tr>
   <td colspan="2">
         [7]<input type="file" name="file" size="15" />
   </td>
     </tr>
 <tr>
   <td>
     [8]<input type="checkbox" name="peripherals" value="burner" id="cbperiferalsburner" />
    <label for="cbperiferalsburner">DVD Writer</label>      <br />
         [8]<input type="checkbox" name="peripherals" value="printer" id="cbperfiferalsprinter"/>
            <label for="cbperfiferalsprinter">Printer</label>       <br />
         [8]<input type="checkbox" name="peripherals" value="cardReader" id="cbperferalscardReader" />
    <label for="cbperferalscardReader">Card Reader </label> <br />
   </td>
   <td>
     [9]<label> <input type="radio" name="browser" value="ff" />
            Firefox
    </label> <br />
 [9]<label> <input type="radio" name="browser" value="ie" />
            Internet Explorer
    </label> <br />
     [9]<label> <input type="radio" name="browser" value="Other" />
             Other
     </label>
   </td>
 </tr>
 <tr>
   <td>
     My Hobbies [10]: <br />
 <select name="hobbies" size="3" multiple="multiple">
    <option selected="selected" value="programming">Hacking JavaScript</option>
<option value="surfing">Surfing the Web</option>
<option value="caffeine">Drinking Coffee</option>
<option value="annoying">Annoying my friends</option>
<optgroup label="unrealistic">
   <option value="dating"> Dating </option>
   <option value="sex">Having Sex </option>
</optgroup>
 </select>
   </td>
   <td>
     My Favorite Color: <br />
 [11]
 <select name="color">
   <option value="green">Green</option>
   <option value="red">Red</option>
   <option value="blue">Blue</option>
 </select>
   </td>
 </tr>
   </table>
 </form>

 <script type="text/javascript">
           addhandlers(document.magicalform);
 </script>
</body>
</html>
```

Every form element has a **name** property that must be set in its HTML tag if it is to be submitted to a server-side program. The form tag itself also has a **name** attribute that you can set. This attribute has nothing to do with form submission, it only exists for the convenience of JavaScript programmers. If the name attribute is defined in a form tag, when the Form object is created for that form, it is stored as an element in the forms[] array of the Document object, as usual, and it is also stored in its own personal property of the Document object. The name of this newly defined property is the value of the name attribute. Furthermore, using a form name makes your code position-independent: it works even if the document is rearranged so that forms appear in a different order.

The tags **img** and **applet** also have name attributes that work the same way as the name attribute of **form**.

When you give a form element a name, you create a new property of the From object that refers to that element. The name of this property is the value of the attribute name. When several elements in a form share the same name (for example, checkboxes and radio buttons), JavaScript simply places those elements in an array with the specified name. The elements appear in the array in the same order in which they appear in the document.

Form element properties:

- **type:** A read-only string that identifies the type of the form element.
- **form:** A read-only reference to the form object in which the element is contained.
- **name:** A read-only string corresponding to the HTML name attribute.
- **value:** A read/write string that specifies the value contained or represented by the form element. This is the string that is sent to the web

server when the form is submitted. What value does the value property hold? It depends on the form element...

  - Button: Text displayed within the button.
  - Radio buttons and checkboxes: Value attributed defined of the corresponding tag.
  - Text and Textarea elements: the text inserted by the user.

Form element event handlers:

- **onclick:** Triggered when the user clicks the mouse on the element. This handler is particularly useful for button and related form elements.
- **onchange:** Trigerred when the user changes the value represented by the element by entering text or selecting an option, for example. Obviously, buttons do not support this event because they do not have an editable value. Note that this event is not triggered every time a user types in a text field, for example. It is triggered only when the user changes the value of an element and then moves the input focus to some other form element.
- **onfocus:** Triggered when the form element receives the input focus.
- **onblur:** Triggered when the form element looses the input focus.

As was already said, many times actually, within the code of an event handle, the this keyword refers to the document element that triggered the event. Since all form elements have a **form** property, the event handlers of a form can always refer to the form object using this.form. Therefore, you can refer to a sibling form element named x using **this.form.x**.

Form elements:

Push Buttons The button element provides a clear visual way for the user to trigger some scripted action. The button element has no default behavior of its own and it is only useful if has an onclick event handler. The value property of the button element controls the text that appears within the button itself. Hyperlinks (anchor elements) provide the same onclick event handler that buttons do and thus any button object can be replaced with a link that does exactly the same thing. Use a button when you want an element that looks like a graphical push button and use a link when the action triggered by the onclick event can be conceptualized as ”follow this link”. Reset and Submit buttons are just like any other buttons except that they have **default actions** associated with them. If the onclick event returns false, the default action of these buttons is not performed. You can use the onclick handler of a submit eleemnt to perform form validation, but it is more common to perform Form validation using the **onsubmit** handler of the Form object itself.

You can also create buttons using the **button** tag, instead of the traditional **input** tag. The button tag is more flexible because instead of displaying the plain text specified in the value attribute, it displays any HTML content. The button tag may be used anywhere in an HTML document, it does not need to be placed within a form. The main difference betweend buttons defined with the button tag and buttons defined with the input tag is that the button tag does not use its value attribute to define its appearance (so you can't change its appearance by setting the value property).

Toogle Buttons The **checkboxes** and **radio buttons** are called **toggle buttons**. That is, buttons that have two clearly distinct states: they can be checked or unchecked. The user can change the state of a toggle button by clicking on it. Radio buttons are meant to be used in groups of related elements, all of which share the same name attribute. Radio elements created in this way are mutually exclusive. Checkboxes are also often used in groups that share the same name attribute. You must remember that when you refer to these kind of element by name you are actually referring an array of same-named elements. Radio buttons and checkboxes define a **checked** property that specifies if the element is currently checked. The **default-Checked** property is a boolean that has the value of the HTML checked attribute, it specifies whether the element is checked when the page is first loaded.
Radio buttons and checkboxes do not display any text themselves. So, setting their value property only changes the string that is sent to the server when the form is submitted.
When the user clicks on a toggle button the onclick and onchange events are triggered.

Text Fields The **text** element allows the user to enter a short, single-line string of text. The value porperty represents the text the user has entered. You can set this property to specify the text that should be displayed in the field. The **textarea** is similar but it allows the user to input multiline text. The **password** element is a modified Text element that displays asterisks as the user types into it. Note that the value of a password element will not be encrypted before submitting the form. That is, it will be sent unencrypted over the network. The **file** element allows the user to enter the name of a file to be uploaded to the web server. The value property of a file element is read-only. This prevents malicious JavaScript programs from tricking the user into uploading a file that should not be shared.

Select The **select** element represents a set of options (represented by Option elements) from which the user can select. This elements are typically rendered as drop-down menus or listboxes. The select element can operate in two distinct ways depending on the value of the **multiple** attribute:

- **multpile="multiple"** The user is allowed to select multiple options and the type property of the Select object is **select-multiple**.
- A single item may be selected and the property is **select-one**.

Options are specified in HTML with the **option** tag, and they represented in JavaScript by Option object stored in the **options**[] array of the Select element. Because a Select element represents a set of choices it does not have a value property. Instead, each Option object contained by the Select object defines a value property.
When the user selects or deselects an object, the Select element triggers the **onchange** event handler. The **selectedIndex** property (read/write) specifies the index of the currently selected Option object. In addition to the **selected** property, the Option object defines a value property (read/write) that specifies the text to be sent to the server when the form is submitted and a **text** property that specifies the string of plain text that appears in the Select element for that option. Note that the Option element does not define form related event handlers; use the **onchange** property of the containing Select element instead.
You can also change the Options array dynamically:

- You can truncate the array (setting the value of options.length).
- You can remove an individual Option (setting its position in the array to null).
- You can create a new Option using the Option constructor. The arguments of the constructor are: the text property (string), the value property (string), the defaultSelected property (bool) and the selected property (bool).

You can use the **optgroup** tag to group options within a Select element. The optgroup tag has a label attribute that specifies text to appear in the Select element. Despite its visual presence, however, an optgroup tag is not selectable by the user, and objects corresponding to the optgroup on the optgroup tag never appear in the options array.

Hidden **Hidden Elements** have no visual representation in a form. It exists to allow arbitrary text to be transmitted to the server when the form is submitted. Sever-side programs use this as a way to save state information that is passed back to them with form submission. Since they have no visual appearance hidden elements cannot generate events and have no event handlers. The **value** property allows you read and write information associated with a hidden element.

```
<input type="hidden" id="age" name="age" value="23" />
```

Fieldset The **fielset** and **label** tags may be used inside HTML forms. These tags are only important for web designers. Placing a fieldset tag within a form causes its corresponding object to be added to the form's elements[] array. Be careful with a code you write when you loop through form elements because this tag does not have the type property!!!!!

For a complete (and very elegant) form validation procedure check page 452 of the book.

## 2.8 Cookies

Aside from being used for server-side scripting, *cookies* are used to implement client-side persistence. All modern browser support cookies and allow them to be scripted using the **Document.cookie.property** object. A *cookie* is a small amount of named data stored by the web browser and associated with a particular web page. Cookies serve to give the web browser a memory, so that it can:

- Use data that was input on one page on another page
- Recall user preferences when the user leaves a page and then returns
- ...

Cookies are implemented as an extension of the HTTP protocol. They are automatically sent to the server, so server-side scripts can read/write cookie values that are stored on the client.

**cookie** is a string property that allows you to read, write, modify and delete the cookie or cookies that apply to the current web page. When you read the value of **document.cookie**, you get a string that contains all the names and values of the cookies that apply to the current document.

Each cookie has several optional attributes that control its lifetime, visibility and security. Cookies are transient by default; the values they hold are stored during the web-browser session but are lost when the user exits the web-browser. To store a cookie you must either:

- Set the **expires** attribute to the expiration date of the cookie(old way to do it).
- Set the **max-age** attribute to the lifetime of the cookie (in seconds)

These cause the browser to store the cookie in a file (it will eventually delete it).

Another important attribute - **path**. This attribute specifies the web page with which the cookie is associated. By default a cookie is associated with the web page that created it and any other web pages in the same directory or subdirectories. By default, cookies are accessible only to pages on the same web server from they were set. Large web sites may want to be shared across multiple web servers, however. For example, the server at **order.example.com** may need to read a cookie value set from **catalog.example.com**. It must set its path to "/" and its **domain** attribute to "example.com". In doing that, it guarantees that the cookie is available to all web pages on **example.com** (and thus to **catalog.example.com** and **orders.example.com**). Note that you cannot set the domain of a cookie to a domain other than the domain of your server.

The cookie attribute **secure** specifies how cookie attributes are transmitted over the network. By default, cookies are insecure. If a cookie is marked secure, it is transmitted only when the browser and the server are connected via HTTPS or another secure protocol.

Note that **expires**, **max-age**, **domain** and **secure** attributes of a cookie are not JavaScript object properties.

To check if cookies are enabled, you can use **navigator.cookieEnabled**. This is not a standard property.

The following example illustrates the use of cookies:

```
<html>
<head>
<script type = "text/javascript">
  function salvaCookie(){
var val = document.formcookie.cookievalue.value;
alert("vou salvar a cookie: " + val);
document.cookie = "cookieval=" + encodeURIComponent(val) +
          "; max-age = " + (24*60*60);
  }

  function mostraCookie(){
    var allcookies = document.cookie;
if(!allcookies){
  alert('nao havia cookies');
  return;
}
    alert('havia cookies');
    var pos = allcookies.indexOf('cookieval=');
    if(pos != -1){
   var start = pos + 10;
   var end = allcookies.indexOf(';', start);
   if(end == -1) end = allcookies.length;
   var value = allcookies.substring(start, end);
   value = decodeURIComponent(value);
   alert('O valor da cookie e: ' + value);
    }
  }

  function checkCookie(){
     if(navigator.cookieEnabled)
   alert("voce pode guardar cookies");
  }
</script>
</head>
<body onload="checkCookie();">
<form name="formcookie">
<label for="cookid">New cookie: </label> <input type="text" id="cookid" name="cookievalue">
<br />
<input type="button" onclick="salvaCookie();" value="Salvar Cookie"/>
<br />
<input type="button" onclick="mostraCookie();" value="Mostrar Cookie" />
```

```
</form>
</body>
</html>
```

Comments on the example:

- We must use **decodeURIComponent()** and **encodeURIComponent()** because cookie values may not include semicolons, commas, or whitespaces.
- You could also have set the path, the domain or the secure attribute of the cookie in the same way we did for the max-age attribute.
- To change the value of a cookie, set its value again using the same name, path and domain along the new value. You can change the lifetime of a cookie when you change its value specifying a new value for the max-age attribute.
- To delete a cookie, set it again using the same name, path, and domain, specifying an arbitrary or empty value, and a max-age attribute of 0. Note that the browser is not required to delete expired cookies immediately, so a cookie may remain in the browser's cookie file past its expiration date.

Cookie limitations:

- Cookies are intended for infrequent storage of small amounts of data. They are not intended as a general purpose communication or data-transfer mechanism.
- The standard does not require the browser to retain more than 300 cookies, 4KB per cookie, 20 cookies per web server.
- Even when cookies are only used for client-side scripting, they are still uploaded to the web server in the request for any web page with which they are associated. When the cookies are not used on the server it is a waste of bandwidth.

Internet Explorer and Flash Actionscript define their own mechanism for data persistence.

## 2.9 JavaScript and XML

The most important of the Ajax web application architecture is its ability to script HTTP with the XMLHttpRequest object. The **X** in Ajax stands for XML since Ajax uses XML-formatted data.

### 2.9.1 Obtaining XML Documents

After making a request using an XMLHttpRequest object, the **responseXML** property of that object refers to a **Document** object that is the parsed representation of the XML document. However this is not the only way to obtain an XML Document object. You can:

- Create an empty XML document
- Load an XML document from an URL (without using the XMLHttpRequest object)
- Parse an XML document from a string
- Obtain an XML document from an XML data island

The techniques for obtaining XML data are usually browser specific.

In DOM level 2 browsers, you can create an empty XML Document using the method **document.implementation.createDocument()**. However, in IE you must proceed differently, as always...

```
if(document.implementation && document.implementation.createDocument)
  return document.implementation.createDocument(namespaceURL, rootTagName, null);
else
  var doc = new ActiveXObject("MSXML2.DOMDocument");
```

When dealing with IE you have to do more things, check the book: page 504.

You can load XML documents from web servers using the XMLHttpRequest object. This is the best way to do it. However, if you don't want to do it that way, you can use the **load()**:

```
XML.load = function(url) {
              var xmldoc = XML.newDocument();
              xmldoc.async = false;
              xmldoc.load(url);
              return xmldoc;
           }
```

We can also use the **load()** function assynchronously:

```
XML.loadAsync = function(url, callback) {
                   var xmldoc = XML.newDocument();
                   xmldoc.onload = function() { callback(xmldoc); };
                   xmldoc.load(url);
                }
```

The **load()** method differs from the XMLHttpRequest in several ways:

- It only works with XML documents, whereas the XMLHttpRequest can be used to download any kind of text document.

- It is not restricted to HTTP protocol. It can be used to read files from the local filesystem.
- When used with HTTP, it generates only GET requests and cannot be used to POST data to a web server.

You can also parse an XML Document from a String of text using a **DOM-Parser** object.

```
XML.parse = function(text) {
  if(typeof DOMParser != "undefined") {
    return (new DOMParser()).parseFromString(text, "application/xml");
  } else if(typeof ActiveXObject != "undefined"){
    var doc = XML.newDocument();
    doc.loadXML(text);
    return doc;
  } else {
    var url = "data:text/xml;charset=utf-8,"+encodeURIComponent(text);
    var request = new XMLHttpRequest();
    request.open("GET", url, false);
    request.send(null);
    return request.responseXML;
  }
```

Microsoft has extended HTML with an ⟨*xml*⟩ that creates an XML data island within the surrounding sea of HTML markup. When IE encounters this ⟨*xml*⟩ tag, it treats it contents as a separate XML document, which you can retrieve using document.getElementById() or other HTML DOM methods. If the ⟨*xml*⟩ tag has a src attribute, the XML document is loaded from the URL specified by that attribute instead of being parsed from the content of the ⟨*xml*⟩ tag.

If a web application requires XML data and the data is known when the application is first loaded, there is an advantage to including that data directly within the HTML page: avoid making new connections to download the data.

Can we simulate data islands in other browsers? Yes we can!!!!

```
XML.getDataIsland = function(id) {
   var doc;
   doc = document.getElementById(id);
   var url = doc.getAttribute('src');
   if(url) doc = XML.load(url);
   else{
     if(!doc.documentElement){
       //This is not already a document
       var docelt = doc.firstChild;
       while(docelt != null) {
         if(docelt.nodeType == Node.ELEMENT_NODE) break;
         else docelt = docelt.nextSibling;
       }
```

```
        doc = XML.newDocument;
        if(docelt) doc.appendChild(doc.importNode(docelt, true));
      }
    }
    return doc;
```

The HTML standard requires browsers to parse tags that they don't know, like ⟨*xml*⟩. This means that browser don't discard XML data within a xml tag. It also means that text included within a data island is displayed by default. To prevent it:

```
<style type="text/css"> xml { display: none; } </style>
```

### 2.9.2 XML DOM

The HTML DOM is an extension of the XML DOM. Therefore, the interfaces HTMLDocument and HTMLElement are extensions of the Document and Element interfaces respectively. What are the differences between DOM Document and DOM Element interfaces and and the corresponding DOM HTML interfaces????

- The method getElementById() always returns null when applied to XML documents.
- The HTML DOM Document interface includes a property body that refers the ⟨*body*⟩ tag within the document. In XML the documentElement property refers to the top level element of the document. This top-level element is also available through the childNodes[] property of the document, but it may not be the only or even the first element of that array because the XML document may contain a DOCTYPE declaration, comments and processing instructions.
- There are several distinctions between the DOM Element interface and the respective HTML DOM interface. Normally, the HTML attributes are made available as properties of the HTMLElement interface. For example, the src attribute of an ⟨*img*⟩ tag is made available as a property of the HTMLElement corresponding to the image. This is not the case in XML DOM: the Element interface has only a single ⟨*tagName*⟩ property. The attributes of an XML element must be explicitly queried and set with getAttribute() and setAttribute(), and related methods.