

**ACADEMIA DE STUDII ECONOMICE - București**  
**Academy Of Economic Studies - Bucharest**

**FACULTY OF BUSINESS ADMINISTRATION**  
(Facultatea de Administrare a Afacerilor cu predare în limbi străine)

**Internet Technologies for Business**

-  
**JavaScripts**

**By: Professor Vasile AVRAM, PhD**  
- suport de curs destinat studenților de la secția engleză -  
(course notes for 1<sup>st</sup> year students of English division)  
- anul I - Zi -

**București 2007**



COPYRIGHT® 2006-2009  
All rights reserved to the author Vasile AVRAM.

# JavaScript

## CONTENTS

Chapter 5 JavaScript.....	4
5.1 JavaScript – An introduction.....	4
5.2 Using and placing JavaScripts in a HTML page.....	8
5.2.1 JavaScript in the body of the HTML file .....	8
5.2.2 JavaScript in heading .....	9
5.2.3 External JavaScripts .....	10
5.3 Defining and using variables.....	10
5.4 Methods.....	11
5.5 Document Object Model (DOM) .....	13
5.6 Using and Defining Function .....	14
5.7 Assignments and expressions .....	16
5.7.1 Arithmetic Expression.....	16
5.7.2 Logical Expression.....	17
5.7.3 String Expression .....	18
5.8 Conditional Execution.....	19
5.9 Popup Boxes.....	23
5.10 Cycles.....	24
5.11 Using events to trigger script execution.....	27
5.12 Handling errors.....	29
References .....	30

## Chapter 5 JavaScript

### 5.1 JavaScript – An introduction

JavaScript is a scripting language that gives HTML designers a programming tool and that can be used for easy management of user interface: it can put dynamic text into a HTML page, it can make the page react to events or it can create and easy manipulate cookies. A JavaScript inserted in the HTML document allows a local recognition and processing (that means at client level) of the events generated by the user such as those generated when the user scans the document or for management of fill-in forms, for example, we must recuperate the information referencing the client (name, address, payment etc). By inserting a JavaScript in the HTML page we can validate the data filled by the client (for example we can validate the Credit Card Account, solvability, transactions history, etc) before it is submitted to the server.

JavaScript allows restructuring an entire HTML document for which we can add, remove, change, or reorder items on a page. In order to change anything on a page, JavaScript needs access to all elements in the HTML document. This access, along with methods and properties to add, move, change, or remove HTML elements, is given through the Document Object Model (DOM).

In 1998, W3C published the Level 1 DOM specification. This specification allowed access to and manipulation of every single element in an HTML page. All browsers have implemented this recommendation, and therefore, incompatibility problems in the DOM have almost disappeared. The DOM can be used by JavaScript to read and change HTML, XHTML, and XML documents. The DOM is separated into different parts (Core, XML, and HTML) and different levels (DOM Level 1/2/3):

- Core DOM - defines a standard set of objects for any structured document;
- XML DOM - defines a standard set of objects for XML documents;
- HTML DOM - defines a standard set of objects for HTML documents.

Every object can have his own Collections, Attributes (Properties) and Methods. Table #.1 shows the JavaScript objects and table #.2 shows the HTML DOM objects.

Table #.1 The JavaScript objects

Object	Description
Window	The top level object in the JavaScript hierarchy. The Window object represents a browser window. A Window object is created automatically with every instance of a <body> or <frameset> tag
Navigator	Contains information about the client's browser
Screen	Contains information about the client's display screen
History	Contains the visited URLs in the browser window
Location	Contains information about the current URL

Table #.2 HTML DOM objects

Object	Description
Document	Represents the entire HTML document and can be used to access all elements in a page
Anchor	Represents an <a> element

Area	Represents an <area> element inside an image-map
Base	Represents a <base> element
Body	Represents the <body> element
Button	Represents a <button> element
Event	Represents the state of an event
Form	Represents a <form> element
Frame	Represents a <frame> element
Frameset	Represents a <frameset> element
Iframe	Represents an <iframe> element
Image	Represents an <img> element
Input button	Represents a button in an HTML form
Input checkbox	Represents a checkbox in an HTML form
Input file	Represents a fileupload in an HTML form
Input hidden	Represents a hidden field in an HTML form
Input password	Represents a password field in an HTML form
Input radio	Represents a radio button in an HTML form
Input reset	Represents a reset button in an HTML form
Input submit	Represents a submit button in an HTML form
Input text	Represents a text-input field in an HTML form
Link	Represents a <link> element
Meta	Represents a <meta> element
Option	Represents an <option> element
Select	Represents a selection list in an HTML form
Style	Represents an individual style statement
Table	Represents a <table> element
TableData	Represents a <td> element
TableRow	Represents a <tr> element
Textarea	Represents a <textarea> element

JavaScript is hardware and software platform independent. Within a JavaScript inserted in the HTML page we can validate the data supplied by the client (for example, to validate the card account, financial availability, history regarding previous transactions etc.).

For an inserted JavaScript the <script type="text/javascript"> and </script> tags tells where the JavaScript starts and ends:

```
<html>
<body>
<script type="text/javascript">
<!--
... // put here the script body
//-->
</script>
</body>
</html>
```

The properties innerText and innerHTML allow us to access the contents - the code - contained in an object. By manipulating the innerText and innerHTML properties, we can change, dynamically, the text on a page (without reloading the page). For example, given a paragraph whose id = "sampleparagraph", its innerText and innerHTML may be accessed via:

document.getElementById('sampleparagraph').innerHTML – this is interpreted as HTML

document.getElementById('sampleparagraph').innerText – this is interpreted as text

If the content of sampleparagraph is "<b> inner text</b>" then:

- innerText would display as **<b>inner text</b>**
- innerHTML would display as **inner text**.

Figure 3.1 shows a HTML form containing a VBScript and a JavaScript.

**Figure 3.1 A Java Script Example**

```
<html>
<head>
<meta name="GENERATOR" content="Microsoft FrontPage 5.0">
<meta name="ProgId" content="FrontPage.Editor.Document">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>New Page 2</title>
<script type="text/javascript" language="javascript">
  <!--
    function calc(a, b){ return (a*b);}
  // -->
</script>
<script id=clientEventHandlersVBS language=vbscript>
  <!--
  Sub Validate_onclick
    document.write("You Type:"+cstr(text1.value)+":"+cstr(text2.value))
  End Sub
  -->
</script>
</head>
<body>
  <script type="text/javascript" language="javascript">
    var welcmess="Welcome to scripts:";
    document.write(welcmess)
  </script>
  <p></p> First &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& Number: <INPUT type="text" ID=Text1
value="0" name="text1" size="20">
  <p></p> Second Number: <INPUT type="text" ID=Text2 value="0" name="text2"
size="20">
  <p></p>&nbsp;&nbsp;&nbsp;&nbsp;& <p><INPUT type="button" value="Show" id="Validate"></p>
</body>
</html>
```

The Style object represents an individual style statement that can be think as an inline style declaration. The Style object can be accessed from the document or from the elements to which that style is applied. For example, given a form whose id = "form1", its styles may be accessed via:

`document.getElementById('form1').style.property`

where property is one of the many style properties available to a given element. Table #.7 shows some common style properties that we can manipulate.

Table #.7 Style properties

Property	Description
style.background	Sets or retrieves the background picture tiled behind the text and graphics in the object.
style.backgroundAttachment	Sets or retrieves how the background image is attached to the object within the document.
style.backgroundColor	Sets or retrieves the color behind the content of the object.
style.backgroundImage	Sets or retrieves the background image of the object.
style.border	Sets or retrieves the width of the border to draw around the object.
style.borderBottom	Sets or retrieves the properties of the bottom border of the object
style.borderBottomColor	Sets or retrieves the color of the bottom border of the object.
style.borderBottomStyle	Sets or retrieves the style of the bottom border of the object.
style.borderBottomWidth	Sets or retrieves the width of the bottom border of the object.
style.borderCollapse	Sets or retrieves a value that indicates whether the row and cell borders of a table are joined in a single border or detached as in standard HTML.
style.borderColor	Sets or retrieves the border color of the object.
style.borderLeft	Sets or retrieves the properties of the left border of the object
style.borderLeftColor	Sets or retrieves the color of the left border of the object.
style.borderLeftStyle	Sets or retrieves the style of the left border of the object
style.borderLeftWidth	Sets or retrieves the width of the left border of the object.
style.borderRight	Sets or retrieves the properties of the right border of the object.
style.borderRightColor	Sets or retrieves the color of the right border of the object.
style.borderRightStyle	Sets or retrieves the style of the right border of the object.
style.borderRightWidth	Sets or retrieves the width of the right border of the object.
style.borderStyle	Sets or retrieves the style of the left, right, top, and bottom borders of the object
style.borderTop	Sets or retrieves the properties of the top border of the object.
style.borderTopColor	Sets or retrieves the color of the top border of the object.
style.borderTopStyle	Sets or retrieves the style of the top border of the object.
style.borderTopWidth	Sets or retrieves the width of the top border of the object.
style.borderWidth	Sets or retrieves the width of the left, right, top, and bottom borders of the object.
style.bottomMargin	Sets or retrieves the bottom margin of the entire body of the page.
style.color	Sets or retrieves the color of the text of the object
style.font	Sets or retrieves a combination of separate font properties of the object. Alternatively, sets or retrieves one or more of six user-preference fonts.
style.fontFamily	Sets or retrieves the name of the font used for text in the object.
style.fontSize	Sets or retrieves a value that indicates the font size used for text in the object.
style.fontStyle	Sets or retrieves the font style of the object as italic, normal, or oblique.
style.fontVariant	Sets or retrieves whether the text of the object is in small capital letters.
style.fontWeight	Sets or retrieves the weight of the font of the object
style.margin	Sets or retrieves the width of the top, right, bottom, and left margins of the object.
style.marginBottom	Sets or retrieves the height of the bottom margin of the object.
style.marginHeight	Sets or retrieves the top and bottom margin heights before displaying the text in a frame.

style.marginLeft	Sets or retrieves the width of the left margin of the object.
style.marginRight	Sets or retrieves the width of the right margin of the object.
style.marginTop	Sets or retrieves the height of the top margin of the object.
style.marginWidth	Sets or retrieves the left and right margin widths before displaying the text in a frame.
style.padding	Sets or retrieves the amount of space to insert between the object and its margin or, if there is a border, between the object and its border
style.paddingBottom	Sets or retrieves the amount of space to insert between the bottom border of the object and the content.
style.paddingLeft	Sets or retrieves the amount of space to insert between the left border of the object and the content.
style.paddingRight	Sets or retrieves the amount of space to insert between the right border of the object and the content.
style.paddingTop	Sets or retrieves the amount of space to insert between the top border of the object and the content.
style.position	Sets or retrieves the type of positioning used for the object.
style.textAlign	Sets or retrieves whether the text in the object is left-aligned, right-aligned, centered, or justified.
style.textDecoration	Sets or retrieves a value that indicates whether the text in the object has blink, line-through, overline, or underline decorations.
style.textIndent	Sets or retrieves the indentation of the first line of text in the object.
style.topMargin	Sets or retrieves the margin for the top of the page.
style.vAlign	Sets or retrieves how text and other content are vertically aligned within the object that contains them.
style.visibility	Sets or retrieves whether the content of the object is displayed.
style.zIndex	Sets or retrieves the stacking order of positioned objects.

The JavaScript sentences involving text strings can be brake up within the text string by using the \ (backslash) character.

The multiline comments can be defined between /\* and \*/; the one line or the inline comments can be defined by using // (two slashes). The extraspace is ignored and the sentences are case sensitive. The ; (semicolon) ending sentence character is optional for sentences defined alone on a line and compulsory for separating the commnds defined in the same line (generally the inline scripts).

## 5.2 Using and placing JavaScripts in a HTML page

In the following paragraphs are introduced some examples of using (and placing) JavaScripts in a HTML page.

### 5.2.1 JavaScript in the body of the HTML file

Java script in the body of the HTML page will be executed when the page loads. Generally, the scripts in the body, will generate the content of the page.

Example:

```
<html>
<head>
  <title>
    Page containing JavaScript
  </title>
</head>
<body>
```



```
<script type="text/javascript">
  document.write("This text is displayed by a JavaScript!")
</script>
</body>
</html>
```

Comments:

The tag <script> have the attribute „type”, that specifies the script type (JavaScript in our case). This script composed by a single command that displays inside the page the text: "This text is displayed by a JavaScript!". If you want include many commands on the same line this must be separated by the „;” (semi colon) character.

The concatenation of text string is realized by using the + character, for example the expression "This text is " + "concatenated." will produce the string "This text is concatenated."

The „/” have a special meaning for the HTML language and consequently when we want display the slash character itself we must precede (prefix) this by a „\” (backslash), as illustrated in this example:

```
document.write("<i>"+"The Operator + is Concatention!"+"</i>")
```

If the script is included in a comment tag (<!--) then the browsers that do not „know” (interpret) JavaScript will not display in the page the script text (script source), as shown in the following sample:

```
<!--
document.write("<i>"+"This text is displayed by a JavaScript!"+"</i>")
//-->
```

The browsers that know JavaScript will process the script, even this included in a comment line.

The string „/”, comment in JavaScript, tell to the browser do not process the line „-->”. We can not use the syntax „/!-”, because a browser that do not interpret JavaScript will display that string „/”.

## 5.2.2 JavaScript in heading

If we want be shure that the script executes before displaying any element in the page we can include this in the heading part of the HTML page (file). The JavaScript in the head section will be executed when called, or when an event is triggered.

Example:

```
<html>
<head>
  <title>
    Page with JavaScript
  </title>
  <script type="text/javascript">
    document.write("This text is displayed by a JavaScript!")
  </script>
</head>
<body>
  <P> This text must appear in the page after the execution of the Javascript.
</body>
</html>
```

The number of scripts placed in the head section and in the body of a HTML page is unlimited.

### 5.2.3 External JavaScripts

A JavaScript can be stored into an external script file from where we can use in many Web pages. In that way the script is written only once and in every HTML file we want use is enough to invoke the file containing the script. The stored script cannot contain the tag `<script>` or his pair `</script>`.

The steps followed when using externaly stored scripts are:

1. The creation of the external file containing the script lines, for example the line:  
`document.write("Text from an external stored script.")`
2. The file is saved with the wanted name and the extension `js` (java script), for example we name the file `scriptex.js`
3. In the HTML pages we want include the stored script file is added the following script:

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

The „src” attribute of the tag `<script>` allows specifying the file containing the script we want execute.

### 5.3 Defining and using variables

JavaScript can contain variable definitions and references to that variables. The variables can be used to store values and the references to that values can be done by referencing the name of the variable. The lifetime of variables can be:

- for variables declared within a function - can only be accessed within the function; they created when encountered their declaration as the function progreses and destroyed when exiting; they called local variables and you can use the same name in diffrent functions;
- for variables declared outside a function – can be accessed anywhere in the page; the name must be unique at that level; the lifetime of these variables starts when they are declared, and ends when the page is closed.

The variable declaration can include an assignment and can be done using one of sentences:

```
var variableName=somevalue
```

or

```
variableName=somevalue
```

In the following example, on define a variable called „mess” that is initialized with the value „This text contained by the variable called mess”, and later on referenced in a write sentence:

```
<html>
<head>
  <title>
    Page with JavaScript
  </title>
</head>
<body>
  <script type="text/javascript">
    <!--
    var mess= " This text contained by the variable called mess"
```

```

document.write("<i>" + mess + "</i>")
//-->
</script>
</body>
</html>

```

The variables, functions and, objects names are case-sensitive and must begin with a letter or an \_ (underscore) character.

## 5.4 Methods

JavaScript is an object based programming language and uses objects (as shown in tables #.1 and #.2). It has many built-in objects such as Area, Image, Date, Window, and Document, and allow also user defining his own objects. In the following table some methods for document and window explained:

Method	Explanation-Example
<b>document.write("msg")</b>	Displays the message „msg” in the page containing the script Example: 1) displaying text in a page: document.write("This text will be displayed in the page") 2) displaying attributes of a page, such as title and URL: <script type="text/javascript"> document.write(document.title+"."+document.URL) </script> </body> *) The formatting of the message to be displayed by write and alert methods or other intrinsic functions that manipulate strings is realized by intermediate of escape sequences.
<b>window.alert("msg")</b>	Displays a dialog box (alert box) containing the message „msg” and the OK button. Example: function display_alert() { alert("The message formatting is ensured" + '\n' + "by using a lot of so called 'escape sequences'") }
<b>window.prompt("msg","default")</b>	Displays a dialog box prompting the user for input and confirm/cancel the dialog. Example: function display_prompt() { var name=prompt("Type your name here","") if (name!=null && name!="") { document.write("You typed " + name + "! It is that correct ?") } }
<b>window.confirm("msg")</b>	Displays a dialog box with a message, a Cancel, and an OK button (similar to MsgBox, ).

	<p>Example:</p> <pre>function display_confirm() {   var buttonpressed=confirm("Press a button")   if (buttonpressed ==true)   {     document.write("You pressed the OK button!")   }   else   {     document.write("You pressed the Cancel button!")   } }</pre>
<b>window.open("URL", "name_of_new_window", "specifications")</b>	<p>Opens a new browser window for the page indicated by the URL argument. The window can be referenced by the name "name_of_new_window" and can be customized by the values supplied by the "specifications" argument.</p> <p>Example:</p> <pre>&lt;html&gt; &lt;head&gt; &lt;script type="text/javascript"&gt; function open_win_ase() {   window.open("http://www.ase.ro","_blank","toolbar=yes,   location=yes, directories=no, status=no, menubar=yes,   scrollbars=yes, resizable=no, copyhistory=yes, width=400,   height=300") } function open_win_avrams() {   window.open("http://www.aavrams.ro","_blank","toolbar=yes,   location=yes, directories=no, status=no, menubar=yes,   scrollbars=yes, resizable=no, copyhistory=yes, width=400,   height=300") } &lt;/script&gt; &lt;/head&gt; &lt;body&gt; &lt;form&gt; &lt;input type="button" value="Faculty" onclick="open_win_ase()"&gt; &lt;input type="button" value="Course Notes" onclick="open_win_avrams()"&gt; &lt;/form&gt; &lt;/body&gt; &lt;/html&gt;</pre>

<sup>\*)</sup> Common Escape Sequences for text display formatting are represented by:

Ampersand	\&
Double quote	\"
Single quote	\'
Newline	\n
Form feed	\f
Carriage return	\r
Backslash	\\
Backspace	\b
Tab	\t

## 5.5 Document Object Model (DOM)

The Document Object Model defines HTML documents as a collection of objects and provides access to every element, identified uniquely by intermediate of an id attribute, in a document. Any element may be accessed (by using the method `getElementById`) and modified by a snippet of JavaScript. The Window object is the top level object in the JavaScript hierarchy (it represents a browser window). A Window object is created automatically with every instance of a `<body>` or `<frameset>` tag. You can see and exercises the various elements of DOM HTML by following the link: [http://www.w3schools.com/html/dom/dom\\_examples.asp](http://www.w3schools.com/html/dom/dom_examples.asp)

Examples:

a) This sample displays the message „This is first paragraph! Click, and see“. If you click somewhere in the displayed text it displays:

„The background is: *the current color name*  
Will be changed in Yellow!“:

```
<html>
<head>
<title>Using DOM</title>
<script language="javascript">
<!--
function xalert()
{
var x=document.getElementById("par1");
x.style.background="red";
alert("The background is:" + x.style.background+"\n Will be changed in Yellow!")
if(x.style.background=="red")
{
x.style.background="yellow";
}
else
{
x.style.background="red";
}
}
-->
</script>
</head>
<body>
<p id="par1" onclick="xalert()" >This is first paragraph! Click, and see</p>
</body>
</html>
```

b) This sample uses innerHTML to change dynamically the header identified by „chgheader“:

```
<html>
<head>
<script type="text/javascript">
function getValue()
{
var x=document.getElementById("myHeader")
alert(x.innerHTML)
}
function chgval()
{
document.getElementById("chgheader").innerHTML="My Header (Changed)"
}
</script>
```

```

</head>
<body>

<h1 id="myHeader" onclick="getValue()">This is first header</h1>
<p>Click on the header to alert its value</p>
<h2 id="chgheader" onclick="chgval()">This is the second header</h2>
<p>Click on the header to change its value</p>

</body>
</html>

```

## 5.6 Using and Defining Function

A function contains code (a set of statements) which is executed when triggered by an event or a call to that function. In JavaScript is possible to use the Java language intrinsic functions or user defined functions (must be defined before any usage).

In the case of user defined functions is preferable that the definition is made in the head section of the HTML page to be shure (or to ensure) in that way they loaded before calling. This required because the browser start processing the HTML page before completely downloading this from server. You may call a function from anywhere within a page (or even from other pages if the function is embeded in an external script).

The general syntax of a function is:

```

function function_name([argument1,argument2,etc])
{
    some_statements
    [return expression]
}

```

where:

*function\_name* is the name the function you want have;

*argument1,argument2,...* the name for the function parameters if it has. Is allowed do not pass any parameter to the function;

*some\_statements* generally variable declarations and executables statements that describes the steps of the algorithm you model. They define together with the return statement (if present) the body of the function;

*expression* is the expression whose evaluation will represent the returned value. A function can return (the sentence return must be present in the body) a value or not (the return statement is not appearing between those of the function's body);

A function can be invoked in one of the ways :

- without arguments:

*function\_name()*

- or with arguments:

*function\_name(argument1,argument2,etc)*

A function is executed when is called. The function can be called within a JavaScript block, via an event handler (inside an HTML tag) or via a href link.

In the following example is called the function „alert” (a standard function of the JavaScript language) with the argument „mess” (the variabe used in the previous example) and that determine the display of an alert box in which the content of variable „mess” displayed.

Example:

The call of an JavaScript intrinsic function.

```

<html>
<head>
<title>

```

```

    Page with JavaScript
  </title>
</head>
<body>
  <script type="text/javascript">
    <!--
    var mess= " This text contained by the variable called mess"
    document.write("<i>"+mess+"</i>") // a method of the document object
    alert(mess) // the intrinsic function invoked
    //-->
  </script>
</body>
</html>

```

Example:

The call of an user defined function.

```

<html>
<head>
  <title>
    Page with JavaScript
  </title>
  <script type="text/javascript">
    <!--
    function suma(a,b)
    {
      result=a+b
      return result
    }
    //-->
  </script>
</head>
<body>
  <script type="text/javascript">
    <!--
    var mess= " This text contained by the variable called mess"
    document.write("<i>"+mess+"</i>")
    alert(mess)
    alert(suma("This text is a ","<String Concatenation>"))
    document.write("This digit is the result of adding 2 to 7 by using the user defined function
    suma:"+suma(2,7))
    //-->
  </script>
</body>
</html>

```

Example :

In this example the JavaScript defined in the head part of the page and in the body part in a href tag.

```

<html>
<head>
<title>A Page with scripts</title>
<script type="text/javascript">
  function chgbgcolor()
  {
    document.bgColor="green"
  }
  function chcolor()
  {
    document.bgColor="orange"
  }

```

```

</script>
<script id=clientEventHandlersJS language=javascript>
<!--
function window_onload() {
    document.bgcolor="orange"
}
//-->
</script>
<script language=javascript for=window event=onload>
<!--
return window_onload()
//-->
</script>
</head>
<body>
<p>The page not empty </p>
<a href="javascript:chcolor(); alert('The background was changed!')">This is a script in
href</a></p>
<h1 id="header1" onclick="chgbgcolor()"> IT4B: </h1>
<h2 id="header2">Errata</h2>
</body>
</html>

```

## 5.7 Assignments and expressions

The general syntax for assignment is:

*variable = expression*

The interpretation of this is: the *variable* before the assignment operator is assigned a value of the *expression* after it, and in the process, the previous value of *variable* is destroyed. An *expression* can be an arithmetic expression, a logical expression or expression on character string. It can be a variable, a constant, a literal, or a combination of these connected by appropriate operators.

An assignment statement stores a literal value or a computational result in a variable and is used to perform most arithmetic operation in a program.

### 5.7.1 Arithmetic Expression

An *arithmetic expression* uses the syntax:

*<operand<sub>1</sub>><arithmetic\_operator><operand<sub>2</sub>>*

where:

- *<operand<sub>1</sub>>*, *<operand<sub>2</sub>>* can be numeric variables, constants or arithmetic expressions (or calls to functions that returns numeric values)
- *arithmetic\_operator* (binary operators) is one of those shown in table #.3.

Table #.3 Arithmetic Operators

Operator	Description	Example Suppose x=2
+	Addition	x+2 = 4
-	Subtraction	5-x = 3
*	Multiplication	x*5 = 10
/	Division	9/3 = 3; 5/2 = 2.5
%	Modulus (division remainder)	5%2 = 1; x%2 = 0; 10%5 = 0
++	Increment By One <sup>1)</sup>	x++ = 3
--	Decrement By One <sup>1)</sup>	x-- = 1



The increment and decrement operators can either be used as a pre- or a post-operator:

**Post-Increment:** the line of code is executed as then the increment/decrement is performed.  
 $y = x++$  is equivalent to the sequence:  
 $y = x$   
 $x = x + 1$

**Pre-Increment:** the increment or decrement is performed before whatever other operations are present within the given line of code.  
 $y = ++x$  is equivalent to the sequence:  
 $x = x + 1$   
 $y = x$

For arithmetic expressions the assignment operator can be combined with the arithmetic ones to define compact expressions as shown in the table #.4.

Table #.4 Assignment Operators

Operator	Example	Is The Same As
=	$x = y$	$x = y$
+=	$x += y$ $x += y - 12$	$x = x + y$ $x = x + (y - 12)$
-=	$x -= y$ $x -= y + 12$	$x = x - y$ $x = x - (y + 12)$
*=	$x *= y$ $x *= 3 + y$	$x = x * y$ $x = x * (3 + y)$
/=	$x /= y$ $x /= y + 2$	$x = x / y$ $x = x / (y + 2)$
%=	$x \% = y$ $x \% = y + 2$	$x = x \% y$ $x = x \% (y + 2)$

## 5.7.2 Logical Expression

A logical expression can be:

- **simple**, with the general syntax:

$\langle \text{variable} \rangle [\langle \text{relation\_operator} \rangle \langle \text{variable} \rangle]$

or

$\langle \text{variable} \rangle [\langle \text{relation\_operator} \rangle \langle \text{constant} \rangle]$

$\langle \text{relation\_operator} \rangle ::= <|<=>|>|=|==|!=$

Table #.5 shows and explains the comparison operators.

- **complex**, with the general syntax:

$e_1 \ \&\& \ e_2$  - logical and;

$e_1 \ || \ e_2$  - logical or;

$! \ e_1$  - logical not.

$\langle \text{logical\_expression}_1 \rangle \langle \text{logical\_operator} \rangle \langle \text{logical\_expression}_2 \rangle$

where the *logical\_operator* can be:

&& (and; two ampersand character), || (or; two vertical bar character) as binary operators (connectives);

! (not) as unary operator.

The precedence of evaluation of logical operators is Not, And, Or. The logical operator together with the truth table defining the way they operate and usage examples are shown in table #.6.

Table #.5 Comparison Operators

Operator	Description	Example Suppose x=2
==	is equal to	x == 3 returns false
===	Is equal to (checks for both value and data type)	y="2" x==y returns true x===y return false (x integer; y string)
!=	is not equal	x != 3 returns true
>	is greater than	x > 3 returns false
<	is less than	x < 3 returns true
>=	is greater than or equal to	x >= 3 returns false
<=	is less than or equal to	x <= 3 returns true

Table #.6 Logical Operators

More Logical Operators				
Operator	Description			Example Suppose x=2; y=3
&&	x	y	and	(x < 9 && y > 1) returns true (x < 9 && y < 1) returns false (x > 9 && y > 1) returns false (x > 9 && y < 1) returns false
	T	T	T	
	T	F	F	
	F	T	F	
	F	F	F	
	x	y	or	(x < 9    y > 1) returns true (x < 9    y < 1) returns true (x > 9    y > 1) returns true (x > 9    y < 1) returns false
	T	T	T	
	T	F	F	
	F	T	F	
	F	F	F	
!	x		not	!(x == y) returns true !(x > y) returns false
	F		T	
	T		F	

### 5.7.3 String Expression

An *expression on character string* can be built (in Java but not only) using:

- the concatenation operator: +
- intrinsic functions for extracting substrings from a string variable or string constant

such as:

Right(*string*,*number\_of\_characters*) - extracting substring from the end

Left(*string*,*number\_of\_characters*) - extracting substring from the beginning

- functions that manipulate strings:

Cstr(*expression*) – convert the expression in a character string;

Lcase(*string\_expression*) – convert the string in lower case;

Ltrim(*string*), Rtrim(*string*), Trim(*string*) – eliminates the spaces (trailing) from left (leading blanks), right and, respectively left-right;

Str(*number*) – converts number in string;

Ucase(*string*) – converts string to uppercase.

The code sequence below is string concatenation by using the concatenation operator + example.

```
text1 = "Faculty of"
text2 = "Business Administration"
text3 = text1 + " " + text2
```

The variable text3 now contains the string "Faculty of Business Administration". The concatenation with the space character " " is realized to separate the strings (generally stored with no trailing blanks).

## 5.8 Conditional Execution

A script can include branches (If...Then...Else...) allowing the definition of conditional executions similarly to the example in the following sequence:

```
if (navigator.appName.indexOf("Internet Explorer")!=-1)
{
    alert("The used Navigator is Internet Explorer!")
}
else
{
    alert("The used Navigator is not Internet Explorer!")
}
```

These code sequence will display an alert box containing a different text depending on the type of the used browser in which the page displayed. The conditional expression of the IF sentence searches for the text „Internet Explorer” in the name of the browser (navigator) application. If this text does not appear in the browser name then the function „indexOf” returns the value „-1”. The condition evaluates to True only if the return of that function is not „-1”. The JavaScript IF sentence, „switch” sentence or the conditional operator „?” can be used to define the conditional execution.

**Decision sentences.** The decision sentences are used to model the decision structure and that is used for choosing an alternative (an operation or block of operations) from two possible alternatives. Algorithm steps that select from a choice of actions are called decision steps.

### If ... Then ... Else ...

```
if (condition)
    operation1;
else
    operation2;
```

If one of operations includes a sentences sequence then this sequence will be included in a sentence block:

```
{
    operationi;
}
```

The decision block can be expressed in a natural language as:

- evaluate the expression that defines the logical condition <condition>;
- If the result of evaluation is True
  - Then execute operation<sub>1</sub>
  - Else execute operation<sub>2</sub>;
- continue the execution with the next step in the flow

### If ... Then ...

**if** (*condition*) *operation*;

```
if (condition) {
    operations;
}
```

### if...else if....else statement

This statement allows to select one of many blocks of code to be executed .

```
if (condition1)
{
    code to be executed if condition1 is true
}
else if (condition2)
{
    code to be executed if condition2 is true
}
else
{
    code to be executed if condition1 and condition2 are not true
}
```

The logical condition  $\langle condition_x \rangle$  is a logical expression that will be evaluated either to True or either to False. The logical conditions can be simple or complex logical conditions.

A simple logical condition has the general syntax:

$\langle variable \rangle$  [ $\langle relation\_operator \rangle$   $\langle variable \rangle$ ]

or

$\langle variable \rangle$  [ $\langle relation\_operator \rangle$   $\langle constant \rangle$ ]

The *relation\_operator* can be one of:

Relation Operator	Interpretation
<	<b>Less than. Example:</b> delta < 0
<=	<b>Less than or equal. Example:</b> delta <= 0
>	<b>Greater than. Example:</b> delta > 0
>=	<b>Greater than or equal. Example:</b> delta >= 0
==	<b>Equal to. Example:</b> a == 0
!=	<b>Not equal. Example:</b> a!=0

The simple logical conditions will be connected by the **AND**, **OR**, and **NOT** logical operators to form complex conditions. The logical operators are evaluated in the order NOT, AND, and OR. The change of the natural order of evaluation can be done by using parenthesis in the same way for arithmetic expressions.

Example:

```
<html> <head> <title>New Page 1</title> </head><body>
<script type="text/javascript">
// If the time is less than 10,write a "Good morning" greeting
// If time between 10 and 16 write a "Good day" greeting
// Otherwise "Hello world"
// Write the hour and If time <12 write AM else write PM
var computerdate = new Date()
var time = computerdate.getHours()
```

```

if (time<10)
{
document.write("<b>Good morning! Now is " +time+((time<12)?' AM':' PM')+"</b>")
}
else if (time>10 && time<16)
{
document.write("<b>Good day! Now is " +time+((time<12)?' AM':' PM')+"</b>")
}
else
{
document.write("<b>Hello World! Now is " +time+((time<12)?' AM':' PM')+"</b>")
}
</script> </body>
</html>

```

**Switch.** Execute one of several groups of statements depending on the value of an expression (called selector). The case structure (and statement) can be especially used when selection is based on the value of a single variable or a simple expression (called the case selector).

```

switch (expression_int) {
    case constant_expression1:
        operations1
    case constant_expression2:
        operations2
    :
    default:
        operationsn
}

```

- *expression\_int* is an expression that must produce an integral value (int);

- *constant\_expression<sub>i</sub>* must be a constant expression;

- the label **default:** can be used only once.

The *expression\_int* is also called the selector of instruction Case.

- if the value of the selector doesn't fit to a constant the operations specified on branch Default (otherwise) will be executed;

- the values of constants must be unique for a *switch sentence*.

Example:

The sequence below uses the switch statement to find out the Romanian name for the day of the week of a date.

```

<HTML>
<HEAD>
<meta name=vs_defaultClientScript content="JavaScript">
<TITLE></TITLE>
<META NAME="GENERATOR" Content="Microsoft Visual Studio">
<META HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8">
<script type="text/javascript">
/* The sequence will write the name of the day in romanian
   (Sunday=0, Monday=1, Tuesday=2, etc)
*/
function RODay(aDayNumber)
{
    switch (aDayNumber)
    {
        case 0:
            return "Duminica"
        case 1:

```

```

        return "Luni"
    case 2:
        return "Marti"
    case 3:
        return "Miercuri"
    case 4:
        return "Joi"
    case 5:
        return "Vineri"
    case 6:
        return "Sambata"
    default:
        alert("What day is it? \n The computer is virused or hardware damaged !")
        return "What day is it? \n The computer is virused or hardware damaged !"
    }
}
</script>
<script id=clientEventHandlersJS language=javascript>
<!--

function Button1_onclick() {
    var i=0
    var datadeazi=new Date()
    var ziua=datadeazi.getDay()
    for (i=ziua;ziua<=6;ziua++){
        document.write(ziua+": " + RODay(ziua)+"<br />")
    }
}

//-->
</script>
<script language=javascript for=Button1 event=onclick>
<!--

return Button1_onclick()
//-->
</script>
</HEAD>
<BODY>
<p>This page contains a Java Script exploiting the switch sentence.</p>
<p>
    <input id=Button1 type=button value="Press This"></p>

</BODY>
</HTML>

```

### Conditional Operator (?)

The conditional operator has the syntax:

*(conditional\_expression) ? true\_case\_expression: false\_case\_expression*

where:

*<conditional\_expression>* is a logical expression that will be evaluated either to True or either to False.

Is a very good idea to include the expression in parenthesis (to enforce his evaluation).

*<true\_case\_expression>* is the expression whose evaluation will be returned if the conditional expression evaluates to True

*<false\_case\_expression>* is the expression whose evaluation will be returned if the conditional expression evaluates to False.

Example:

```



<HTML>
<HEAD>
<meta name=vs_defaultClientScript content="JavaScript">
<TITLE></TITLE>
<META NAME="GENERATOR" Content="Microsoft Visual Studio">
<META HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8">
</HEAD>
<BODY>

<script language=javascript>
var TotalBalance, savings=300
TotalBalance =(savings==0) ? 0:(savings*1.03)
// TotalBalance is now 309
document.write("Total Balance is now: " + TotalBalance)
</script>
</BODY>
</HTML>

```

## 5.9 Popup Boxes

In JavaScript we can create three kinds of popup boxes by invoking the associated intrinsic function. These functions are:

Function	Description
alert("text_to_be_displayed")	<p>Displays an alert box containing the message passed in argument and an OK button. This call produces the box:</p> 
confirm("text_to_be_displayed")	<p>Displays a box containing the message passed in argument provided with an OK (confirm) and Cancel (deny) button. This call produces the box:</p> 
prompt("text_to_be_displayed", "defaultvalue")	<p>Displays an input dialog box provided with a text box to fill data and an OK (return the value typed to the caller) and Cancel (return a Null value to caller). This call produces the box:</p>



## 5.10 Cycles

The repeating structure repeats a block of statements while a condition is True or Until a condition becomes True. The repetition of steps in a program is called a **loop**.

The repeated execution of a sequence of instructions (loop) can be done by using the looping sentences „while” , „do...while” and, for.

Example:

```
<html>
<head>
<title>
  Page containing loop
</title>
</head>
<body>
<script type="text/javascript">
<!--
  for (i=0; i<5; i++)
  {
    document.write("Step i: "+i+"<br>")
  }
-->
</script>
</body>
</html>
```

In this example the values of i are written into the page „Step i:the\_vale\_of\_i” from the value 0 to 4.

### a) The condition evaluated first

- first syntax:

**while** (*condition*) *operation*;

- second syntax:

```
while (condition)
{
  operations;
  [continue;]
  [break;]
}
```

where:

- **continue** jump to the condition evaluation;
- **break** interrupt the cycle and transfer the execution to the sentence that follows to the end block marker }



*Note. The ; character ending sentences is optional if the sentence is written alone on the line. It is necessary if you define mode sentences on the same line.*

The executions of such blocks follow the scenario (while): the condition is evaluated and if the condition evaluates to:

- **True** then executes the block of statements;
- **False** then end the execution of the cycle (Loop) and continue the execution of the program.

If for the first time the condition is False the sentence block is simply skipped.

#### **b) the condition evaluated after**

```
do
{
    operations;
} while conditions
```

In this case the operation is executed first and then the condition is evaluated and can be described as:

- the *operations* are executed;
- the *condition* is evaluated;
- **if** the result of the evaluation of the *condition* is False *then loop* to execute again the *operations*;
- **if** the evaluation of the *condition* is True *then* continue the execution of the program (and close the loop).

#### **c) counted loop**

```
for (expression1; expression2; expression3) operation;
```

If many operations desired in the cycle they must be included as block;

*expression<sub>1</sub>* – is an expression that initializes the counter, having the general syntax *counter=startvalue*;

*expression<sub>2</sub>* – contains the definition for ending the loop, generally a logical condition of the form *counter<=endvalue*;

*expression<sub>3</sub>* – is an expression to increment or decrement the value for the counter, for example *counter=counter+increment*.

The cycle can be unconditionally stopped by using the instruction **break** and can be unconditionally restarted by using the sentence: **continue**.

Example:

```
for (counter = iv; fv; s) {operations};
```

The execution of For sentence follows the scenario:

1. The value *i<sub>v</sub>* is assigned to the variable *counter*;
2. The value of variable *counter* is compared with the end value *f<sub>v</sub>* (the value can be determined by evaluating an expression);
3. The operations are executed;
4. The value of variable *counter* is incremented with the value step (1 if step not specified);
5. Repeat the steps from 2 to 5.

Example :

Figure 3.# shows the usage of document object for accessing the forms collection and displaying, as comma separated values, the attributes Name, Value and Type.

```

<html>
<head>
<title>A form and javascript</title>
</head>
<body>

<form method="POST" action="--WEBBOT-SELF--">
  <!--webbot bot="SaveResults" u-
file="C:\Documents and Settings\Vio\My
Documents\My Webs\_private\form_results.csv"
s-format="TEXT/CSV" s-label-fields="TRUE" -->

  <!-- This is the description of the form -->
  <p>First name:<input type="text"
name="FName" size="20"></p>
  <p>Last Name:<input type="text"
name="LName" size="20"></p>
  <p>Gender:<input type="radio" value="V1"
name="Male" checked>Male
  <input type="radio" name="Female" value="V2">Female</p>
  <p><input type="submit" value="Submit" name="B1">
  <input type="reset" value="Reset" name="B2"></p>

</form>

<script type="text/vbscript">
  document.write("Name, Value, Type "+<br />")
</script>
<script type="text/javascript">
  for (i=0; i < document.forms[0].elements.length;i++)
  {
    document.write(document.forms[0].elements[i].name + ", ");
    //syntax below uses the name attribute of the form to access the form's elements
    document.write(document.forms[0].elements[i].value + ", ");
    document.write(document.forms[0].elements[i].type + "<br />");
  }
</script>
</body>

</html>

```



Figure 3.1 Accessing HTML form elements

### For ... In statement

**for** (*variable in object*)

```

{
  code to be executed
}

```

Example :

In this example is defined an array object called divisions and the first three elements initialized. The for...in sentence will fill in the HTML document the lines initialized in the array.

```

html>
<body>

```

```

<script type="text/javascript">

var x, nr

var divisions = new Array()
divisions[0] = "English"
divisions[1] = "French"
divisions[2] = "German"

for (x in divisions)
{
nr=x/1+1;
document.write(nr+": "+divisions[x] + "<br />")
}
</script>

</body>
</html>
that produces the output :
1: English
2: French
3: German

```

## 5.11 Using events to trigger script execution

Some events that can be associated with HTML pages are represented by the following:

Event	Occurs when...
onabort	a user aborts page loading
onblur	a user leaves an object
onchange	a user changes the value of an object
onclick	a user clicks on an object
ondblclick	a user double-clicks on an object
onerror	an error occurs
onfocus	a user makes an object active
onkeydown	a keyboard key is on its way down
onkeypress	a keyboard key is pressed
onkeyup	a keyboard key is released
onload	a page is finished loading (in Netscape, this event occurs during the loading of a page).
onmousedown	a user presses a mouse-button
onmousemove	a cursor moves on an object
onmouseover	a cursor moves over an object
onmouseout	a cursor moves off an object
onmouseup	a user releases a mouse-button
onreset	a user resets a form
onselect	a user selects content on a page
onsubmit	a user submits a form

onunload	a user closes a page; a frequent usage is to deal with cookies.
----------	---

The table below shows common usage of events:

Event	Usage
onload, onunload	The onload event is often used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information. Both events frequently used to deal with cookies.
onfocus, onblur, onchange	Generally used in combination with validation of form fields.
onsubmit	Is used to validate All form fields before submission (is possible to deal with logical validation involving more fields from the form).
onmouseover, onmouseout	Generally used for creating "animated" buttons.

In the following example is shown an inline JavaScript code (without the tags `<script>` and `</script>`). The web page contains a button whose property „Caption” has the value „ASE”. When the event „onclick” occurs (when clicking the button) is called the function „open” (member of „windows” functions group), having in arguments the arguments required to open the ASE site home page in a window called internally „ase\_home”.

Example:

```
<html>
<head>
<title>
  Command button link
</title>
</head>
<body>
  <form>
    <input type="button" value="ASE" onclick='window.open("http://www.ase.ro", "ase_home")'>
  </form>
</body>
</html>
```

Example:

```
<html>
<head>
<title>Pagina cu Cronometru </title>
<script type="text/javascript">
  function startTime()
  {
    var today=new Date()
    var h=today.getHours()
    var m=today.getMinutes()
    var s=today.getSeconds()
    // add a zero in front of numbers<10
    m=checkTime(m)
    s=checkTime(s)
    document.getElementById('txt').innerHTML=h+":"+m+":"+s
    t=setTimeout('startTime()',500)
  }

  function checkTime(i)
  {
    if (i<10)
    {i="0" + i}
```

```

        return i
    }
</script>
</head>

<body onload="startTime()">
<div id="txt" align="right"></div>
<script type="text/javascript">
    var x, nr
    var divisions = new Array()
    divisions[0] = "English"
    divisions[1] = "French"
    divisions[2] = "German"
    for (x in divisions)
    {
        nr=x/1+1;
        document.write(nr+": "+divisions[x] + "<br />")
    }
</script>
</body>
</html>

```

## 5.12 Handling errors

In JavaScript are two ways for catching errors in a Web page:

- by using the try...catch statement;
- by using the onerror event.

### Try...catch

The code you want prevent harassing user by error messages is included between try sentence and catch(err) sentence:

```

try
{
    //Run some code here
}
catch(err)
{
    //Handle errors here
}

```

### Throw

Throw statement allows user to create an exception that can be catch and processed later on by try..catch. The syntax is:

```
throw(exception)
```

### onerror event

The syntax for the onerror event and his associated error handler is:

```
Onerror=handleError
```

```

function handleError(msg, url, l)
{
    // handle the error here
    return true (success) or false (failure)
}

```

## References

1.	[AvDg03]	Vasile Avram, Gheorghe Dodescu	Informatics: Computer Hardware and Programming in Visual Basic, Ed. Economică, București, 2003 (Chp. 1.6, 1.7, 1.8, 7.11.3 and 7.11.4)
2.	[DgAv05]	Gheorghe Dodescu, Vasile Avram	Informatics: Operating Systems and Application Software, Ed. Economică, București, 2005 (Chp. 10.1, 10.2 and 10.3)
3.	[BIS-TDM]	Dave Chaffey, Paul Bocij, Andrew Greasley, Simon Hickie	Business Information Systems-Technology, Development and Management for the e-business, Prentice Hall, London, second edition, 2003
4.	[BF01]	Benjamin Faraggi	Architectures marcanes et portails B to B, Ed. DUNOD, Paris, 2001
5.	[RFC 1630]	T. Berners-Lee	RFC 1630 - Universal Resource Identifiers in WWW, Network Working Group, CERN, June 1994
6.	[RFC3986]	T. Berners-Lee W3C/MIT, R. Fielding Day Software, L. Masinter Adobe Systems	Uniform Resource Identifier (URI): Generic Syntax, January 2005
7.	[KLJL]	Kenneth C. Laudon, Jane P. Laudon	Essentials of Management Information Systems – Managing the Digital Firm, Prentice Hall, fifth edition, 2003
8.	[W3C]	www.w3c.org	World Wide Web Consortium, Web standards collection
9.	[MNSS]	Todd Miller, Matthew L. Nelson, Stella Ying Shen and Michael J. Shaw	e-Business Management Models: A Services Perspective and Case Studies, Revere Group
10.	E-commerce business models <a href="http://www.iusmentis.com">http://www.iusmentis.com</a> <a href="http://www.iusmentis.com/business/ecommerce/businessmodels/">http://www.iusmentis.com/business/ecommerce/businessmodels/</a>		
11.	<a href="http://digitalenterprise.org/models/models.html">http://digitalenterprise.org/models/models.html</a> Professor Michael Rappa, North Carolina State University		