

# **Client-Side Programming with JavaScript**

© Riccardo Sisto, Politecnico di Torino

# What is JavaScript?

- A cross-platform object-oriented scripting language for web clients (and servers)
- Originally developed by Netscape (LiveScript, 1995) to add interactivity to HTML pages
- Interpreted language (browsers include JS interpreters)
  - ⇒ Source Portability
  - ⇒ Performance not as good as with compiled languages (but this is not very relevant on clients)
- Quite different from Java
  - the commonality with Java stands mainly in the C-like syntax and in some features inspired by Java

# Java and JavaScript

## Java

- **Compiled** to bytecode
- Object-oriented, based on classes and instances
- Inheritance based on **class hierarchy**
- Classes are defined at compile-time. No type change at runtime
- Variable type **must** be declared
- Restrictions apply (e.g. for writing to disk) **in applets**

## JavaScript

- **Interpreted** in source form
- Object-oriented, based on classes and instances
- Inheritance based on the **prototype** mechanism
- Methods and attributes can be dynamically added at runtime
- Variables **need not be declared**
- Restrictions apply (e.g. for writing to disk) **in any script**

# JavaScript, Jscript, ECMAScript

- Some dialects of JavaScript exist
  - most notably, **Jscript** (Microsoft) and **ActionScript** (Adobe)
- The core of JavaScript, common to the other dialects, has been standardized by ECMA (European Computer Manufacturers Association) as **ECMA-262**
- The standard language has been renamed ECMAScript
- The standard has been issues also as ISO/IEC-16262
- JavaScript (Netscape) JScript (IE) and ActionScript all conform to the standard
  - each of them adds nonstandard extensions and a host context (objects for interacting with the environment)

# References

- These slides refer to JavaScript 1.8.X (complying with ECMA 262 5.1)
- Documentation:
  - Core JavaScript Guide  
<https://developer.mozilla.org/en/JavaScript/Guide>
  - JavaScript Reference Guide  
<https://developer.mozilla.org/en/JavaScript/Reference>
  - ECMAScript Standard  
<http://www.ecma-international.org/publications/standards/Ecma262.htm>

# What can we do with JavaScript?

- HTML cannot make decisions, iterate tasks, react to single keystrokes or mouse clicks,...
- Web scripting languages like JavaScript combine scripting with HTML in order to make pages more **interactive**
- In particular, with JavaScript you can make pages that
  - react to events (e.g. a user clicks on an HTML element, loading a page terminates, a user hits a key on the keyboard,...)
  - read and modify any HTML element of the displayed document
  - detect the browser being used
  - create and view cookies
  - open new windows, tabs and pop-ups

# Typical Uses of Client-Side Scripts

- **Validation** of form data during form filling
  - saves processing time on the server and makes reaction to errors faster
- **User interactions** not possible with simple forms (e.g. alert boxes for message visualization and data entry)
- **Page animation** cursor movement, image drawing, etc.
- **Adaptation** of HTML pages to the browser
- **And much more** (e.g. contextual menus, messages on the browser status line, opening windows and pop-ups,...)

# Including JavaScript into HTML

- JavaScript code can be included using the HTML **script** element in two different forms:

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

```
    document.write("Hello world!")
```

```
    ...
```

```
</script>
```

JavaScript code

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

```
    src="http://my.site/menu.js">
```

```
</script>
```

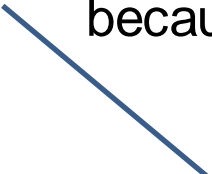
JavaScript filename

- When a browser encounters a script element:
  - interprets the code included in or referenced by the script element (`document.write` statements write in the script position)



# Script Attributes

- **TYPE** : the MIME type (specifies the language the script belongs to)
- **SRC**: a reference to the source file of the script
- **DEFER**: specifies that execution can be deferred because the code does not produce output
- **REQUIRED**



Can improve rendering speed with no-output scripts because the browser has not to wait for script termination

# Alternative Contents

- A `noscript` element can be added after a script element in order to detect cases when JavaScript code cannot be executed.

- Example:

```
<script type="text/javascript">  
    document.write("Hello world!")
```

```
    ...
```

```
</script>
```

```
<noscript>
```

```
    Sorry: Your browser does not support or has  
    disabled javascript
```

```
</noscript>
```

# Old/Nonstandard Browsers

- There is a trick based on JavaScript comments that can be used with old or nonstandard browsers that cannot interpret the script and noscript tags:

```
<script type="text/javascript"><!--  
    document.write("Hello world!")  
    ...  
// --></script>
```

Start comment

One-line comment

End of comment

One-line comment

Script-unaware browser interpretation

Script-aware browser interpretation

# Script Positioning

- Script elements in the HTML **HEAD**
  - Processed when a page loads but **before** starting page rendering
  - Useful for writing META tags or for including code that must be readily available in the whole page (e.g. functions)
- Script elements in the HTML **BODY**
  - Processed when the element **is encountered** by the browser in the page processing flow
  - Useful for writing parts of the HTML body

# Code Execution Time

- When the page loads
  - Script elements in the HEAD
- When the page is displayed
  - Script elements in the BODY
- When invoked by other code fragments or when some environment event occurs
  - functions in any (already processed) script element

# Error Reporting

- Normally JavaScript is executed in a browser
- Each browser has its own way for error reporting
- Examples

Browser	JavaScript Error Reporting
Firefox	Error console displayed by Shift-Ctrl-J (Shift-Command-J)
Chrome	Error console displayed by Shift-Ctrl-J (Shift-Command-J)
IE	Automatic after customization (in <b>Options-&gt;Advanced</b> , uncheck <b>Disable Script Debugging</b> and check <b>Display Notification about Every Script Error</b> )

[js-error.html](#)

# Basic Communication with the Environment (I/O)

- Different I/O targets supported:
  - Browser Window
  - HTML Document that is being displayed
  - Client Keyboard
  - Client Mouse
  - Files
- Pre-defined objects represent environment elements (e.g. `window`, `document`)
- Code can be associated with events (e.g. mouse clicks)

# The Main Forms of I/O

- **`document.write`** writes to the current HTML page
  - The characters written are interpreted by the **HTML** interpreter (formatting tags can be used)
- **`window.alert`** creates an alert window with the specified text (no HTML) and one confirm button
- **`window.confirm`** similar to alert but the window has 2 buttons (one for confirming, one for cancelling) and a boolean is returned
- **`window.prompt`** similar to confirm with an additional input text field and returns the entered string



# Alert Example

```
<html>
<head><title>Hello world (with alert)</title></head>
<body>
<script type="text/javascript"><!--
    alert("Hello world!")
//--></script>
<noscript>
    Sorry: Your browser does not support or has
    disabled javascript
</noscript>
</body>
</html>
```

window is implied

[js-alert.html](#)

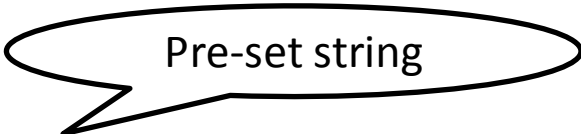
# Confirm Example

```
<html>
<head><title>Confirm Example</title></head>
<body>
<script type="text/javascript"><!--
    if(confirm("press OK to proceed"))
        document.write("<H1>You confirmed</H1>");
    else
        document.write("<H1>You cancelled</H1>");
//--> </script>
<noscript>
    Sorry: Your browser does not support or has
    disabled javascript
</noscript>
</body>
</html>
```

[js-confirm.html](#)

# Prompt Example

```
<html>
<head><title>Confirm Example</title></head>
<body>
<script type="text/javascript"><!--
    val = prompt("Enter location\n", "Torino");
    if(val)
        document.write("<H1>"+"You confirmed: "+val+
                        "</H1>");
    else
        document.write("<H1>You cancelled</H1>");
//--> </script>
...
</body>
</html>
```



Pre-set string

<js-prompt.html>

# Execution Blocking

- **prompt, confirm**
    - Execution **is blocked** waiting for user input
  - **alert**
    - Behavior differs according to OS:
      - UNIX-like systems: Execution is not blocked
      - Windows systems: Execution is blocked
- => If you want to block execution, use prompt or confirm

# Main Language Elements

- Basic Syntax Rules
- Constants and Variables
- Operators and Expressions
- Control Flow Constructs
- Functions
- Arrays
- Objects

# Comments

- **Single-line** comments introduced by `//` or by `<!--`
- **Multi-line** comments delimited by `/*` and `*/`
  - Start of comment `/*`
  - End of comment `*/`

# Statement Termination

- Each statement should be terminated by a *semicolon* (`;`)
- The semicolon may be substituted by a *carriage-return*
- Examples:

`a = 3`

optional

`b = 4;`

mandatory

`a = 3; b = 4;`

- Using the semicolon is a good habit!

# Identifiers

- In JavaScript identifiers are used to name variables, functions, labels, ...
- Identifiers are words such that
  - The initial is a letter, underscore (\_) or dollar (\$)
  - The next characters are letters, digits, underscores or dollars
  - The word is not a reserved word
- Javascript is *case sensitive*
  - *Var1*  $\neq$  *var1*



# Reserved words (1)

<b>abstract</b>	<b>delete</b>	<b>function</b>
<b>boolean</b>	<b>do</b>	<b>goto</b>
<b>break</b>	<b>double</b>	<b>if</b>
<b>byte</b>	<b>else</b>	<b>implements</b>
<b>case</b>	<b>enum</b>	<b>import</b>
<b>catch</b>	<b>export</b>	<b>in</b>
<b>char</b>	<b>extends</b>	<b>instanceof</b>
<b>class</b>	<b>false</b>	<b>int</b>
<b>const</b>	<b>final</b>	<b>interface</b>
<b>continue</b>	<b>finally</b>	<b>long</b>
<b>debugger</b>	<b>float</b>	<b>native</b>
<b>default</b>	<b>for</b>	<b>new</b>

# Reserved words (2)

<b>null</b>	<b>throw</b>
<b>package</b>	<b>throws</b>
<b>private</b>	<b>transient</b>
<b>protected</b>	<b>true</b>
<b>public</b>	<b>try</b>
<b>return</b>	<b>typeof</b>
<b>short</b>	<b>var</b>
<b>static</b>	<b>void</b>
<b>super</b>	<b>volatile</b>
<b>switch</b>	<b>while</b>
<b>synchronized</b>	<b>with</b>
<b>this</b>	

# Literals

- Literals are used to represent constant values.

Examples:

`12`

`1.2`

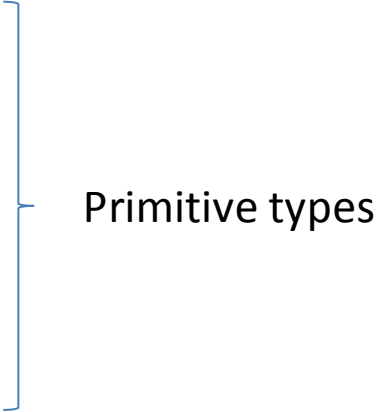
`"ciao" // a string`

`'ciao' // another way of writing  
// a string`

`true, false`

`null`

# Types

- Javascript built-in types:
    - Number
    - Boolean
    - String
    - **null**
    - **Undefined**
    - Object
- Primitive types
- 

# Numbers

- A number is a double precision floating point numeral written in decimal, octal or hexadecimal, with integer or real notation
  - range for integers:  $[-2^{53}, 2^{53}]$
  - least non-zero floating point:  $\pm 5 \times 10^{-324}$
  - greatest floating point:  $\pm 1.7976931348623157 \times 10^{308}$

- decimal numerals start with no extra leading 0 digit

**examples:**    77        0.987645        -3.278e-30

- octal numerals start with 0

**examples:**    077                    02376

- hexadecimal numerals start with 0x or 0X

**examples:**    0x77    0X123456789abcDEF

# Special Number Constants

- **Infinity** or **Number.POSITIVE\_INFINITY**
  - Represents  $+\infty$
- **Number.NEGATIVE\_INFINITY**
  - Represents  $-\infty$
- **NaN** or **Number.NaN**
  - Represents a result that is “not-a-number”
- **Number.MAX\_VALUE**
  - the greatest non-infinite number that can be represented
- **Number.MIN\_VALUE**
  - the least positive non-zero number that can be represented

**Number** is a built-in object

# Booleans and Strings

- Booleans

- There are just two boolean literals: `true` and `false`
- They are automatically converted into 1 and 0 when necessary

- Strings

- Sequences of characters written between `"` or `'`
- examples: `"23"` `'JavaScript'`

# Special Characters

- In addition to normal character symbols and UNICODE characters (`\uxxx`, where `xxx` is an hexadecimal numeral) JavaScript strings may include:
  - `\'` (simple quote)                      `\"` (double quote)
  - `\b` (backspace)                      `\f` (form feed)
  - `\n` (line feed)                      `\r` (carriage return)
  - `\t` (tab)                      `\\` (backslash)
  - `\v` (vertical tab)                      `\0` (NUL)



# Variables

- JavaScript is **loosely** typed
- Variables in JavaScript can be used with much freedom
  - a variable can be identified by any identifier
  - a variable has no fixed type
    - a type is assigned at each assignment
    - type may change according to the context where the variable is referenced
  - a variable needs no declaration
  - a variable can be assigned by the assignment operator =

# Example: Variable Assignments

```
<SCRIPT TYPE="text/javascript">
```

```
<!--
```

```
height=2+3.4;
```

Number type

```
document.write(height);
```

```
document.write("<BR>");
```

```
height="5.4 meters";
```

String type

```
document.write(height);
```

```
// --> </SCRIPT>
```

[js-var-assign.html](#)

# Variable Declaration

- Even if declaration is not necessary ***declaration is possible (and declaring variables is a good practice)***
- Variable declarations are introduced by the `var` keyword:  

```
var a,i,str;
```
- Multiple declarations are legal
- A variable declared **inside a function** is **local**
- A variable used **without declaration** is always **global**

# Variable Initialization

- A declared variable has value `undefined` until a value is assigned to it
- Any attempt to read a variable with undefined value raises a runtime error (just like reading a variable with no declaration and no previous assignment)

# Variable Scoping

- There are just two possible scopes for variables
  - Local
    - variables declared inside a function
  - Global
    - variables declared outside a function
    - variables used without an explicit declaration
- JavaScript has no *block scope*

# Arrays

- Arrays in JavaScript have C-like syntax with integer offsets starting at 0

- Example:

`document.write(a[1])` writes the second element of array `a`

- An array can be simply created by assigning a variable a list of values delimited by square brackets

- Example:

`a = ['x', 'xyz', 'awl']` // `a` now contains an array

# Multidimensional Arrays

- The array syntax can be extended to multiple dimensions by simply having array elements that are arrays
- Examples:

```
a = [  
    ['x','y','z'],  
    ['a','b','c'],  
    ['d','e','f']  
]
```

```
fst = ['x','y','z']
```

```
snd = ['a','b','c']
```

```
trd = ['d','e','f']
```

```
arr = [fst, snd, trd]           // equivalent to a
```

# Objects

- Object = collection of *properties*
  - similar to a C struct**but**
  - properties may contain *values*, nested *objects* or *functions*
  - properties are dynamic (may be created and destroyed at runtime)
- A property inside an object is referenced by the dot notation:
  - `image.height`
  - `image.length`





# Methods and built-in Objects

- A property that is a function is called a **method**
- Note that JavaScript is object-oriented but objects may exist even without classes
- JavaScript has a collection of **built-in objects** that can be referenced globally (and are like libraries):
  - Math, Date, RegExp, JSON
  - Object, Function, Array
  - String, Boolean, Number
  - The global object
  - Error objects: Error, EvalError, RangeError, SyntaxError, ...

# Global Object

- When the JavaScript interpreter starts the execution of a script, a *global object* is created
- Any variable declared or used within the script is considered as a property of the global object.
- Any built-in function is a method of the global object
- The global object can be referenced by **this**
  - Note that inside a function **this** has a different meaning!

# Call object

- The local variables of a function are properties of a special object called **call object**
- A call object is created when a function execution starts
- A call object exists only for the duration of the corresponding function execution

# Constants

- A constant is like a variable but it is read-only
- A constant is such if it is declared using the `const` keyword:
  - `const buf_len=81;`
  - `const month="february";`
- Some constants are properties of the predefined object **Math**
  - Example: `Math.PI` denotes  $\pi$

# Expressions and Operators

- Expressions can be built combining literals, constants, variables, operators and function calls (C-like syntax)
- Simple expressions (no operator)  
`"result"`  
`3.2`
- Unary operators  
`-2.3`
- Binary operators  
`3.14159*2.564`
- Ternary operators  
`val>5? 4:0;`

# Arithmetic Operators

Operator	Example	Meaning
+	<b>x + y</b>	addition
-	<b>x - y</b>	subtraction
*	<b>x * y</b>	multiplication
/	<b>x / y</b>	division, returns a double
++	<b>++x</b>	pre-increment
	<b>x++</b>	post-increment
--	<b>--x</b>	pre-decrement
	<b>x--</b>	post-decrement
-	<b>-x</b>	opposite
%	<b>x % y</b>	modulus (division reminder)

# Relational Operators

- Return a boolean value

<b>&gt;</b>	<b>x &gt; y</b>	greater than
<b>&gt;=</b>	<b>x &gt;= y</b>	greater than or equal to
<b>&lt;</b>	<b>x &lt; y</b>	less than
<b>&lt;=</b>	<b>x &lt;= y</b>	less than or equal to
<b>==</b>	<b>x == y</b>	is equal to
<b>!=</b>	<b>x != y</b>	is not equal to
<b>===</b>	<b>x === y</b>	is equal to (and of same type)
<b>!==</b>	<b>x !== y</b>	is not equal to (and of same type)
<b>in</b>	<b>a in x</b>	is a property of object
<b>instanceof</b>	<b>y instanceof x</b>	is an instance of prototype

# Equality Operators

- **`x==y`**
  - True if x and y are the same, **after data type conversions**
- **`x===y`**
  - True if x and y are the same both in **value**, and **type**
- Example
  - `var x=3.2, y="3.2";`
  - `x==y`            **is true**
  - `x===y`          **is false**



# Inequality Operators

- **`x != y`**
  - True if x and y are different, **even after data type conversions**
- **`x !== y`**
  - True if x and y are of different value or type
- Example
  - `var x=3.2, y="3.2";`
  - `x != y`      **is false**
  - `x !== y`     **is true**

# Example of instanceof

```
var birthday=new date(2000, 9, 12);
```

```
if (birthday instanceof date).....
```

  
true

# Logical and Bitwise Operators

<code>&amp;&amp;</code>	<code>x &amp;&amp; y</code>	AND
<code>  </code>	<code>x    y</code>	OR
<code>!</code>	<code>!x</code>	NOT

- Bitwise operators

<code>&amp;</code>	<code>x &amp; y</code>	bitwise AND
<code> </code>	<code>x   y</code>	bitwise OR
<code>^</code>	<code>x ^ y</code>	bitwise EXOR
<code>~</code>	<code>~x</code>	bitwise NOT
<code>&lt;&lt;</code>	<code>x&lt;&lt;3</code>	left shift ( <code>x&lt;&lt;3</code> )
<code>&gt;&gt;</code>	<code>x&gt;&gt;3</code>	right <b>arithmetic</b> shift
<code>&gt;&gt;&gt;</code>	<code>x&gt;&gt;&gt;3</code>	right <b>logical</b> shift

# The Assignment Operator

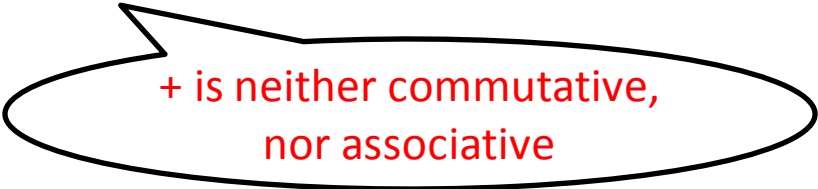
- Assignment is an operator, like in C
- Example:
  - `val=3*10.45;`
  - `x=y=0;`

# Other Assignment Operators

- **`x += y`** is equivalent to **`x = x + y`**
- **`x -= y`** is equivalent to **`x = x - y`**
- **`x *= y`** is equivalent to **`x = x * y`**
- **`x /= y`** is equivalent to **`x = x / y`**
- **`x %= y`** is equivalent to **`x = x % y`**
- **`x <<= y`** is equivalent to **`x = x << y`**
- **`x >>= y`** is equivalent to **`x = x >> y`**
- **`x >>>= y`** is equivalent to **`x = x >>> y`**
- **`x &= y`** is equivalent to **`x = x & y`**
- **`x ^= y`** is equivalent to **`x = x ^ y`**
- **`x |= y`** is equivalent to **`x = x | y`**

# String Operators

- Relational operators can be applied on strings
- Concatenation: `+`                      `+=`
- Examples
  - `"x:" + 3`                      is equivalent to `"x: 3"`
  - `1 + 2 + " times"` is equivalent to `"3 times"`
  - `"x: " + 1 + 2`                is equivalent to `"x: 12"`
  - `name = "Charles"`
  - `name += " Dickens"`



`+` is neither commutative,  
nor associative

# The typeof Operator

- Returns the name of the operand type as a string
- Examples:
  - `typeof 17.54` is equivalent to `"number"`
  - `typeof ("Hello")` is equivalent to `"string"`
  - `typeof true` is equivalent to `"boolean"`
  - `typeof null` is equivalent to `"object"`
  - `typeof parseInt` is equivalent to `"function"`
- If applied to a variable returns its current type (if the variable is not declared or has not yet been assigned returns `undefined`)
- Example
  - ```
if(typeof(x)=="string")  
    document.write("x is a string variable: "+x);
```

# The `void` Operator

- Prefix operator applied to an expression in order to evaluate the expression **without using the result**
- Always returns **undefined** (but any side effect of expression evaluation will be observable)
- Example:

```
<a href="javascript:void(myfun())">
```

```
Click here to call myfun</a>
```

Lets the user invoke `myfun` without effects on the displayed page.



# The Conditional Operator

- Like in C:

$(\text{<condition>})? \text{<expr1>} : \text{<expr2>}$

- Semantics
  - If *condition* is true the result is *expr1*
  - else the result is *expr2*

# The comma Operator

- Like in C:
  - The left operand is evaluated first
  - Then the right operand is evaluated
  - The result of the expression is the result of the evaluation of the right operand
- Example:
  - `i=28,k="hello",n=3.24;`
  - The result is 3.24 (and the assignments are executed as a side effect)

# Other Miscellaneous Operators

- **new**
  - Makes a new instance of an object
- **delete**
  - deletes the property of an object

# Operator Precedence Classes

class	operators	
1	( ) [ ] .	parenthesis, call member
2	++ --	increment/decr
3	+ - ~ !	unary
4	* / %	multiplicative
5	+ -	additive
6	<< >> >>>	shift
7	&	bitwise and
8		bitwise or
9	^	bitwise exor
10	< > >= >=	relational

class	operators	
11	== != === !==	equality
12	&&	logical and
13		logical or
14	? :	ternary cond.
15	= += -= *= /= %= <<= >>= >>>= &= ^=  =	assignment
16	,	sequential eval.

# Operator Associativity

- Right-to-left

`new`

object creation

`++ --`

increment and decrement

`+ - ! ~`

unary operators

`? :`

conditional

`= *= /= %= += ...`

assignment

- Left-to-right

- all the other operators

# Explicit Casting

- In JavaScript there is no cast operator but there are conversion functions
- A string can be converted to a **number** by the functions:
  - `parseInt(str)` for integers
  - `parseFloat(str)` for realsIf the string is not a number, the result is **NaN**
- A string can be converted to a **boolean** by the function:
  - `Boolean(str)`
- Any data can be converted to a **string** by the function
  - `String()`

# Statements

- The simplest statement is a single assignment

**`i=3.2478;`**

- Sequences of statements can be grouped into blocks (delimited by curly braces as in C):

```
{  
    x=3;  
    y="hello";  
    z=x+y;  
}
```

# if Statement

- Syntax (C-like):

`if( <predicate> ) <statement1> else <statement2>`

Optional part



- Examples

```
- if(x<=3) k=0;  
- if(x<=3) k=0 else k=-1;  
- if(x<=3)  
    k=0  
  else if (y==0)  
    k=-1  
  else k=1;
```



# switch Statement

- Syntax (C-like):

```
switch( <expression> ) {  
    case <label1>: <statement_1>;  
    case <label2>: <statement_2>;  
    ...  
    default: <statement_n>;  
}
```

- A label can be **any expression** (even with non-integer and non-numeric value) Examples:
  - `Math.PI`
  - `v[3]`

# for Loop

- Syntax (C-like):

**for** ( *<initialization>* ; *<condition>* ; *<update>* )  
    *<statement>*



- Example:

```
for ( s=0,x=5,y=1 ; x>0 ; x--,y+=2 )  
    s+=x*y;
```

# for.....in Loop

- Syntax:

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

- Example (print properties of object my\_object):

```
for (prop in my_object) {  
  name= "Property Name: " + prop;  
  val = "Value: " + my_object[prop];  
  document.write(name+"-"+val+"<br>");  
}
```

# while Loop

- Syntax (C-like):

**while**( *<condition>* )

*<statement>*

- Example:

```
var x=5;
```

```
var y=1;
```

```
while (x>1)
```

```
  {y*=x; x--};
```

# do...while Loop

- Syntax (C-like):

`do <statement>`

`while( <condition>)`

- Example:

```
var x=5;
```

```
var y=1;
```

```
do {
```

```
    y*=x; x--
```

```
} while (x>1);
```

# break Statement

- Like in C, interrupts the innermost loop or switch

- Example:

```
switch (x){  
    case "item1": y="10, 20, 30";  
                break;  
    case "item2": y="2, 4, 6";  
                break;  
    default: y="no availability of "+x;  
}
```

# Labels and continue

- `continue` causes the control flow to proceed to the next iteration
- A label can be used to identify a particular program point (same C syntax)
- Labels cannot be referenced by `goto` (there is no `goto` in JavaScript) but can be referenced by `continue` and `break`

# Labels and continue Example

```
checkiandj :  
  while (i<4) {  
    document.write(i + "<BR>");  
    i+=1;  
    checkj :  
      while (j>4) {  
        document.write(j + "<BR>");  
        j-=1;  
        if ((j%2)==0)  
          continue checkiandj;  
        document.write(j + " is odd.<BR>");  
      }  
    document.write("i = " + i + "<br>");  
    document.write("j = " + j + "<br>");  
  }
```



# with Statement

- Lets the programmer refer implicitly to a given prefix
- Syntax:

```
with(<object>)
```

```
<statement>
```

- Example

```
x=Math.sin(i*Math.PI/4);
```

```
y=Math.cos(j*Math.PI/4);
```

is equivalent to

```
with(Math){  
    x=sin(i*PI/4);  
    y=cos(j*PI/4;  
};
```

# Functions

- Can be defined by the user in various ways
  - By a function definition
  - By a literal function definition
  - By the `Function` constructor
- A function can be called only if it is built-in or it has already been defined within the script (with a scope including the call site)

# function Definition

Optional



- Syntax (C-like):

```
function <identifier>( <parameter list> )  
{ <function body> }
```

- Example:

```
function nfact(n)  
{ var i=1;  
  while(n>1) {i*=n; n--};  
  return i  
}
```

# Recursion

- Javascript admits recursive functions
- Example:

```
function nfact_rec(n)
{ if (n>1)
    return n*nfact_rec(n-1)
  else return 1
}
```

# Argument Passing

- Function arguments are always passed **by value**

but

- If an object is passed as argument, the object property values can be modified
- Actual argument lists may have **variable length**
  - Arguments passed in excess can be accessed by the built-in array `arguments[i]`
  - The actual number of arguments passed in a call can be accessed by the built-in property `arguments.length`

# Example: Accessing a Variable-Length Argument List

- Example:

```
function myConcat(separator) {  
    result=""  
    for(var i=1; i<arguments.length; i++) {  
        result+=arguments[i]+separator  
    }  
    return result  
}
```

- Call example:

```
myConcat(" ", "1, 2.32, "3", 4)
```

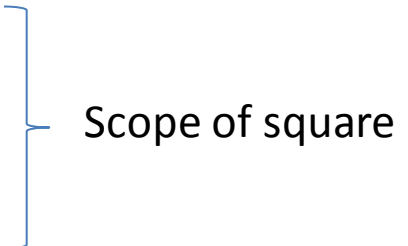
- Output:

```
1, 2.32, 3, 4,
```

# Nested Functions

- A function can be defined inside another function
- Example:

```
function addSquares (a,b) {  
    function square(x) {  
        return x*x  
    }  
    return square(a) + square(b)  
}  
  
a=addSquares(2,3) // a is 13  
b=addSquares(3,4) // b is 25  
c=addSquares(4,5) // c is 41
```



Scope of square

# Scoping Rules

- JavaScript uses **lexical scoping rules**
- A function definition opens a new scope with accessibility to
  - whatever is accessible in the outer scope
  - variables, arguments and functions **defined** in the function block (these are visible only in the new scope and its sub-scopes, except variables used without the var keyword, which are global)
- Example:

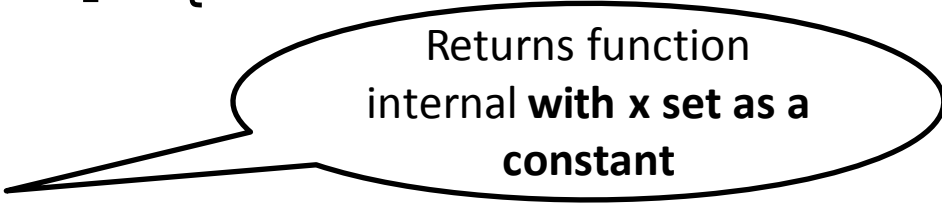
```
var x = 10;
function testScope(y) { return y + x; }
alert(testScope(7)); // displays 17
function testEnv() {
    var x = -1;
    return testScope(7); // displays 17
}
```



# Functions as Objects

- Functions are objects: they can be passed as arguments, returned as return values and assigned to variables
- Example:

```
function external(x) {  
    function internal(y) {  
        return x-y  
    }  
    return internal  
}
```



Returns function  
internal **with x set as a  
constant**

```
result = external(3)(5) // result is -2
```

# Function Expressions (Literal Functions)

- A literal function is a **single-use** function definition
  - Scope is limited to the place where the function is defined
  - Name can be omitted

- Examples:

```
var f = function(x){return x*x};  
var i = 3.24;  
i=f(i);  
obj.prop(function(x){return x*x});  
var hundred = (function(x){return x*x})(10)
```

Stored in variable

Passed as argument

Directly called

# The Function Constructor

- Can be used to define a function “on the fly” (i.e. at runtime)

- Syntax:

`new Function( <argument list> , <function body> )`

Optional



List of  
strings

string

- Example:

```
var square = new Function("x", "return x*x");  
var i=3.22;  
i=square(i);
```

Function call



# More about arguments

- Other properties of arguments
  - `arguments.caller` name of calling function
  - `arguments.callee` name of called function
  - `arguments.callee.length` number of arguments (in the definition)

- Example:

```
function check( param ) {  
    var actual = arguments.length;  
    var expected = arguments.callee.length;  
    if (actual != expected)  
        return false;  
    else  
        return true;  
}
```

- The use of `arguments` as a property of Function is deprecated ( only the variable `arguments` should be used)

# Some Useful built-in (global) Functions

- **`escape(str)`**
  - Encodes `str` using hexadecimal escape sequences for non-ASCII characters so that it can be ported on any platform
- **`unescape(str)`**
  - The inverse of `escape(str)`
- **`isFinite(val)`**
  - Returns true if `val` is (a number) different from `NaN`, `Number.POSITIVE_INFINITY`, `Number.NEGATIVE_INFINITY`
- **`isNaN(val)`**
  - Returns true if `val` is not a number ( `NaN` ).

# Some Useful built-in (global) Functions

- **`eval(str)`**
  - Interprets `str` as a string containing JavaScript code and executes this code.

# More About JavaScript Objects

- Objects exist independently of classes
  - Each object has its own properties and methods
  - Each new object has a fresh copy of properties and methods
- Classes of objects sharing some properties can be created using the object **prototype** (a special built-in property available in every object)

# Object Creation

- Objects can be created by simply listing their properties and initial values as comma separated *name:value* pairs
- Example:

```
var 3D_point = {x:2, y:8.32, z:-2.45};  
i = 3D_point.x;  
j = 3D_point.y;  
k = 3D_point.z;
```
- All properties are public



# Adding and Deleting Properties

- A new property can be **added** by simply assigning it a value
- Example:

```
3D_point.t=30.4 // new property t added
```

- A property can be **removed** by the `delete` operator
- Example:  

```
delete 3D_point.x // the other properties  
                  // remain unaffected
```

# Adding Methods

- This is just a particular case of property addition
- Example:

```
3D_point.dist=function( ) {Math.sqrt(  
this.x*this.x+this.y*this.y+this.z*  
this.z) };
```

↑  
Gives access to the other object properties

- The new method can be invoked **only on the object it has been added to**

# Object Creation Using Constructors

- The object creation method discussed so far requires that properties are specified **each time** an object is created
- A **constructor** is a way to specify properties **once for all objects of a particular type**
- A constructor can be invoked by the **new** operator
  - ⇒ creates a new object with the properties specified by the constructor

# Object Creation Using Constructors

- Example: A constructor for objects representing complex numbers
  - Properties:
    - **re** real part
    - **im** imaginary part
    - **modulus** method that computes the modulus
    - **phase** method that computes the phase

```
function modulus(){
    return Math.sqrt(this.re*this.re+this.im*this.im)
};
function phase(){
    return Math.atan(this.im/this.re)
};
function complex(x,y) {
    this.re = x;
    this.im = y;
    this.modulus = modulus;
    this.phase = phase;
}
```

```
var num= new complex(-2.1,3.7);
i=num.modulus();
j=num.phase();
```

# Other Possible Formulation

```
function complex(x,y) {  
    this.re = x;  
    this.im = y;  
    this.modulus = function() {  
        return Math.sqrt(this.re*this.re+this.im*this.im)  
    };  
    this.phase = function() {  
        return Math.atan(this.im/this.re)  
    };  
}
```

```
var num= new complex(-2.1,3.7);  
i=num.modulus();  
j=num.phase();
```

# Empty Objects

- An empty object (without custom properties) can be created by the Object constructor (built-in):

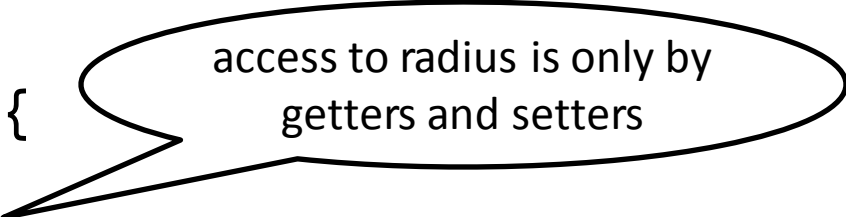
```
var x = new Object();
```

- Properties can be added to an empty object as already shown

# Making Properties Private

- Even if properties are all public, private properties can be simulated by variables declared inside functions
- Example:

```
circle = function(){  
  var radius;  
  this.setRadius = function(x){radius = x};  
  this.getRadius = function(){return radius};  
  this.area = function(){  
    return radius*radius*Math.PI  
  }  
}
```



access to radius is only by  
getters and setters



# Accessing Properties by Expressions

- The name used to reference a property can be **an expression**
- Example:

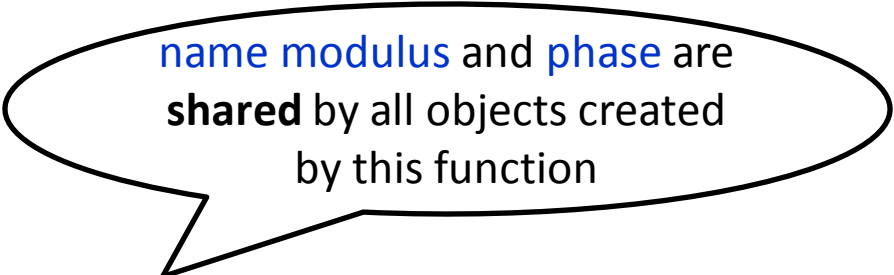
```
var a = {x:-0.33; y:3.14 ;z:6.54};  
i = "z";  
height = a[i]; //access to a.z
```

# Using Prototypes

- The definition of a constructor is similar to a **class** definition but all the properties defined in a constructor (including methods) are like **instance** variables
  - ⇒ each object created with **new** gets a **fresh copy** of each method (and of each other property)
  - ⇒ this may waste memory
- Prototypes enable the equivalent of static (class) variables
- Each function has a **prototype** property that holds properties that must not be replicated when creating new objects using the function as a constructor
  - ⇒ a method defined in the prototype is not replicated

# Example

```
function complex(x,y) {  
    this.re = x;  
    this.im = y;  
}  
  
complex.prototype.name = "complex number";  
complex.prototype.modulus = function() {  
    return Math.sqrt(this.re*this.re+this.im*this.im)  
};  
complex.prototype.phase = function() {  
    return Math.atan(this.im/this.re)  
};
```

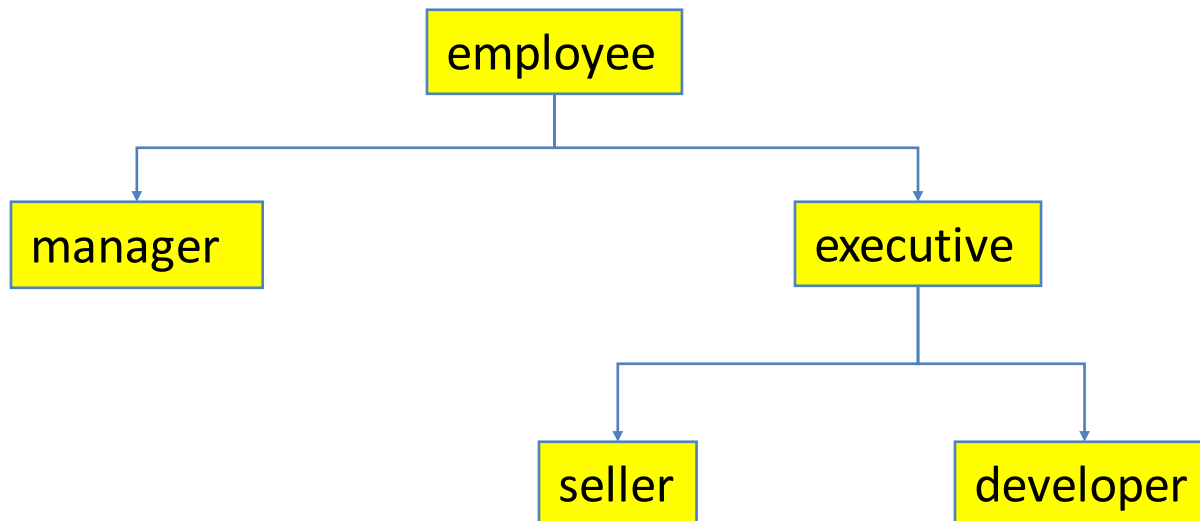


name modulus and phase are shared by all objects created by this function

```
var num= new complex(-2.1,3.7);  
i=num.modulus();  
j=num.phase();
```

# Inheritance

- The *prototype* property can be exploited even for creating new object constructors (i.e. object classes) by inheritance
- Example: create constructors for the following classes of objects bound by inheritance relationships:



# Definitions

## Employee

```
function employee(){  
    this.name="";  
    this.dep="general";  
}
```

## Manager

```
function manager(){  
    this.reports=[];  
}  
manager.prototype=new employee();
```

## Executive

```
function executive(){  
    this.projects=[];  
};  
executive.prototype=new employee();
```

## Seller

```
function seller(){  
    this.dep="sales";  
    this.quote=234;  
}  
seller.prototype=new executive();
```

## Developer

```
function developer(){  
    this.machines=[];  
    this.dep="technical";  
}  
developer.prototype=new executive();
```

# Object Creations

```
var x = new employee()  
// x.name is ""  
// x.dep is "general"  
var y = new manager()  
// y.name is ""  
// y.dep is "general"  
// y.reports is []  
var z = new executive();  
// z.name is ""  
// z.dep is "general"  
// z.projects is []
```

```
var w = new seller();  
// w.name is ""  
// w.dep is "sales"  
// w.projects is []  
// w.quote is 234  
var m = new developer();  
// m.name is ""  
// m.dep is "technical"  
// m.projects is []  
// m.machines is []
```

# Core Objects

- Let us examine the main properties of the built-in core objects:
  - Array
  - Boolean
  - Function
  - Date
  - Math
  - Number
  - Object
  - String
  - RegExp

# Array

- Array objects are **unbounded** collections of elements, even of **different types**, indexed by an integer index

⇒ When creating an array no size nor type are necessary:

```
var arr = new Array();
```

- The current length of an array is given by the **length** property:

```
if(arr.length<=3) ...
```



# Other Constructor Variants

```
var v = new Array(<val0>, ..., <valn-1>);
```

```
var vect = new Array(100);
```

↑  
if this is not a number, it is  
interpreted as the only element

```
var a = new Array();
```

```
a[0]=0; a[99]=100;
```

Array with 100 elements  
only the first and last ones  
are defined. The other ones  
have value **undefined**

# Alternative Constructor Syntax

The following expressions are equivalent:

```
var v = new Array(<val0>, ..., <valn-1>);
```

```
var v = Array(<val0>, ..., <valn-1>);
```

```
var v = [<val0>, ..., <valn-1>];
```

The following expressions are equivalent:

```
var v = new Array();
```

```
var v = Array();
```

```
var v = [];
```

# Constructors with Multidimensional Arrays

- Straightforward because multidimensional arrays are arrays of arrays

- Example:

```
var mat = Array(  
    Array('x','y','z'),  
    Array('a','b','c')  
);
```

# Checking Array Bounds

- Of course bound checking is up to the programmer
- Example:

```
for(var i=0; i<a.length; i++)  
    if(a[i] != undefined) {  
        msg = "Position: " + i + " Value: " + a[i];  
        document.write(msg);  
    }
```

# Array Methods

- **concat**

- Concatenates one array with one or more other arrays:

```
num1=[1,2,3]
num2=[4,5,6]
num3=[7,8,9]
nums=num1.concat(num2,num3)
// nums is [1,2,3,4,5,6,7,8,9]
```

- **join**

- returns a string with the elements of the array separated by the comma default separator:

```
nums.join()    // returns the string
                // "1,2,3,4,5,6,7,8,9"
```

# Array Methods

- **reverse**
  - reverses the elements of the array
- **pop**
  - returns and removes the last element of the array
- **push**
  - appends new element(s) after the last element
  - returns the new size of the array

# Array Methods

- **shift**

- returns and removes element 0, and shifts the other elements back by one position

- **unshift**

- appends element(s) at the head of the array

```
nums.unshift(10,11,12)
```

```
// result is [10,11,12,1,2,3,4,5,6,7,8,9]
```

# Array Methods

- **slice(<start>, <end>)**
  - Creates a new array containing the elements from element **<start>** up to but not including element **<end>**
- **splice(<start>, <len>, <val<sub>0</sub>>, ..., <val<sub>n</sub>>)**
  - the **<len>** elements starting at index **<start>** are replaced by the next arguments **<val<sub>0</sub>>, ..., <val<sub>n</sub>>**
  - if there are no enough arguments, the remaining elements are eliminated (become undefined)
- **sort**
  - sorts the array (as an optional argument it is possible to pass the comparator function to be used for sorting)



# Associative Arrays

- Associative arrays have elements referenced by name rather than by numeric offset
- An associative array can be created by the following syntax:

`{ <key1> : <el1> , ... , <keyn> : <eln> }`

– where `<keyi>` is the i-th key and `<eli>` is the i-th element

- Example:

```
directory = { "7073" : "Riccardo Sisto" ,  
             "1223" : "Mario Rossi" }
```

```
for (k in directory)
```

```
    document.write(k + " : " + directory[k] + "<br/>")
```

# String

- String objects are string containers with string methods
- constant strings are automatically converted to string objects when necessary
- Differences between string constants and objects:  

```
s1 = "2 + 2"  
s2 = new String("2 + 2") // creates String object  
eval(s1) // returns 4  
eval(s2) // returns "2 + 2" (a string)
```
- The length of a string object is given by the `length` property

# Math

- Container for mathematical functions and constants
  - Properties: **mathematical constants**
  - Methods: **mathematical functions**
- **Differently from other core objects, Math cannot be used as a constructor**

# Boolean

- The constructor can be used to create boolean objects from other objects
- Example:

```
var boo = new Boolean(val)
```

- If `val` has any of the values `false`, `0`, `null`, `""` then `boo` takes value `false`  
else `boo` takes value `true`

# RegExp

- A RegExp object represents a **regular expression**
- RegExp objects are typically used for string matching
- A RegExp object can be created with the syntax:

*/ <expr> /*

where *<expr>* is a regular expression. The main syntax is:

expr	matches
.	any single char
<i>element</i> *	<i>element</i> 0 or more times
<i>element</i> +	<i>element</i> 1 or more times
<i>element</i> ?	<i>element</i> 0 or 1 time
[ <i>chars</i> ]	1 char of those in brackets
[^ <i>chars</i> ]	1 char of those not in brackets

expr	matches
<i>left right</i>	either <i>left</i> or <i>right</i>
<i>l-r</i>	range of chars from <i>l</i> to <i>r</i>
\d	single digit
\n	newline
\s	whitespace character
\t	tab character
\w	word character [a-zA-Z0-9_]

# RegExp

- RegExp objects can be used in methods `match`, `search`, `replace` and `split` of `String`

- Example:

```
re = /(\w+)\s(\w+)/;  
str = "First Second";  
newstr=str.replace(re, "$2, $1");  
document.write(newstr)//newstr is "Second First"
```

- RegExp objects have the methods `test`, `exec`

- Example:

```
re = /[A-Z][a-z]*/;  
if (re.test(str))  
    document.write("str is well formed");
```

# Date

- Includes utility functions for date and time manipulation
- Several constructors are available:
  - `Date()`
  - `Date(milliseconds)`
  - `Date(string)`
  - `Date(Y,M,D,H,M,s,ms)`
    - `M` from 1 to 12
    - `D` from 1 to 31
    - `H` from 0 to 23
    - `M` from 0 to 59
    - `s` from 0 to 59
    - `ms` from 0 to 999

# Main Date Methods

- `getDate()`, `setDate()`
- `getFullYear()`, `setFullYear()` (4 digits)
- `getHours()`, `setHours()`
- `getMilliseconds()`, `setMilliseconds()`
- `getMinutes()`, `setMinutes()`
- `getMonth()`, `setMonth()`
- `getSeconds()`, `setSeconds()`
- `getYear()`, `setYear()`



# try...catch Statement

- Errors can be handled by a java-like try-catch statement
- Syntax:

```
try {  
    <block1>  
} catch( <error_var> )  
{ <block2> }
```
- when a runtime error occurs in <block<sub>1</sub>> <error\_var> is bound to the **Error** object that represents the error and <block<sub>2</sub>> is executed
- note that syntax errors do not trigger the catch block

# How to Inspect the Error Object

If `error` is the variable bound to an Error object:

- `error.name` is the error **class**
- `error.message` is the error **description**

# Standard Error Classes

- **EvalError** :
  - error in executing `eval()`
- **RangeError** :
  - a numeric value exceeded its allowable range
- **ReferenceError**
  - an invalid reference value (e.g. undefined variable or function) has been detected
- **SyntaxError**
- **TypeError**
  - the type of an operand is different than the expected type
- **URIError**
  - error when using a URI

# Example

- [js-error-trycatch.html](#)

# Custom Errors

- Errors can be generated by the application using the **throw** statement
- Syntax:  
`throw <error object>`
- An error object of one of the standard classes can be created using the constructor (and defining the message)
- The programmer may want to create custom error classes.

# Example (1)

```
function entrycheck(){
  try{
    var agecheck=prompt("How old are you?")
    if (isNaN(parseInt(agecheck)))
      throw new Error("Please enter a valid age")
    else if (agecheck<13)
      throw new Error("You are too young!")
    else alert("Enjoy!")
  }
  catch(e){
    alert(e.name+" "+e.message)
  }
}
```

# Example (2)

```
<html>
<head>
<script type="text/javascript">
  function entrycheck(){
    ...
  }
</script>
</head>
<body>
  <input type="button" value="Check"
    onclick="entrycheck()">
</body>
</html>
```

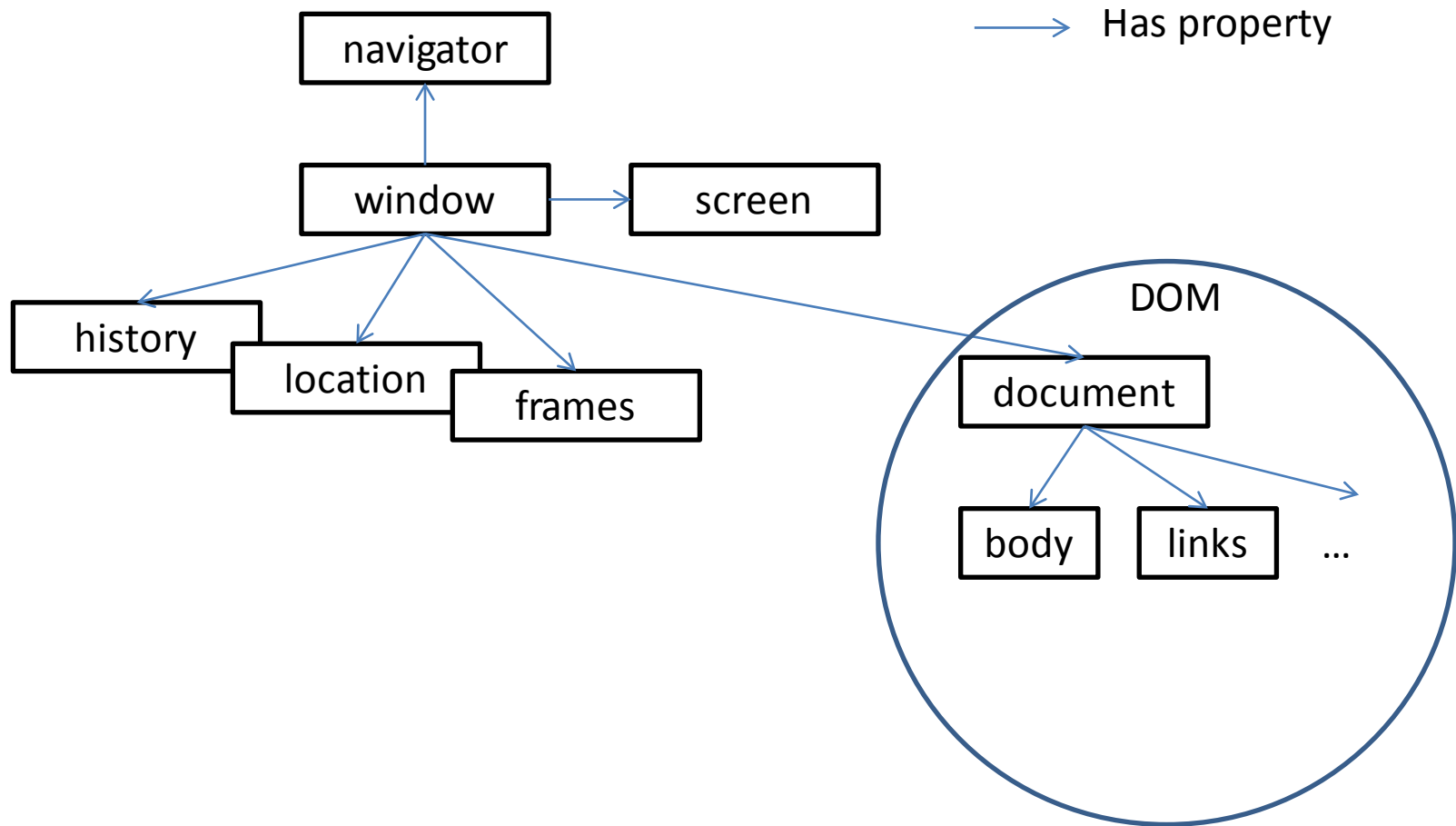
[js-age.html](#)

# The Client Environment (I/O)

- We are going to study the following I/O programming elements for JavaScript in the **client side**:
  - **Environment Objects (e.g. window)**: interface for reading browser information and for controlling the browser
  - **Document Object Model (DOM)**: API for manipulating (read/write) the HTML document that is being displayed in a browser window
  - **Event Programming**: mechanisms for programming reaction to events (e.g. mouse click, mouse double click, mouse pointer entering a certain area, ...)
- **Important: this part is not included in the ECMA standard**
  - We'll study only the features supported by most browsers (look at reference documentation to check browser support)



# The Hierarchy of Environment Javascript Object Properties



# The window Object

- Global object that includes the script being executed
  - Any variable defined in the script also becomes a property of **window**
  - When referencing the **window** object, the name (**window**) can be omitted (is implied)
  - Each **<FRAME>** in the document generates a new **window** object
  - Initially the referenced window object is the one associated with the **<FRAME>** where the script is located
  - There are methods for changing the **window** object being referenced
  - The **<FRAMESET>** that includes the current frame is referenced by the **parent** property.

# The window Object

- Main properties of `window`:
  - `name` the name of this window (assigned by `open( )`)
  - `self`, `parent`, `top`
  - `frames[ ]` all the frames in the window
  - `location` the current URL
  - `history` the history of visited pages
  - `document` the document displayed in the window
  - `status` the text in the status bar of the browser
  - `opener` the name of the window that opened this window

# The window Object

- Main methods of `window`
  - `alert()`, `prompt()`, `confirm()`
  - `open()` open new window (or tab)
  - `close()` close this window
  - `moveBy(dx,dy)` move this window
  - `moveTo(x,y)` move this window
  - `resizeBy(dx,dy)` resize this window
  - `resizeTo(x,y)` resize this window
  - `focus()` give focus to this window
  - `blur()` take focus out of this window
  - `print()` open the printer window

# The open Method

- Returns a reference to the new window (**opener** is a reference back to the opener window)

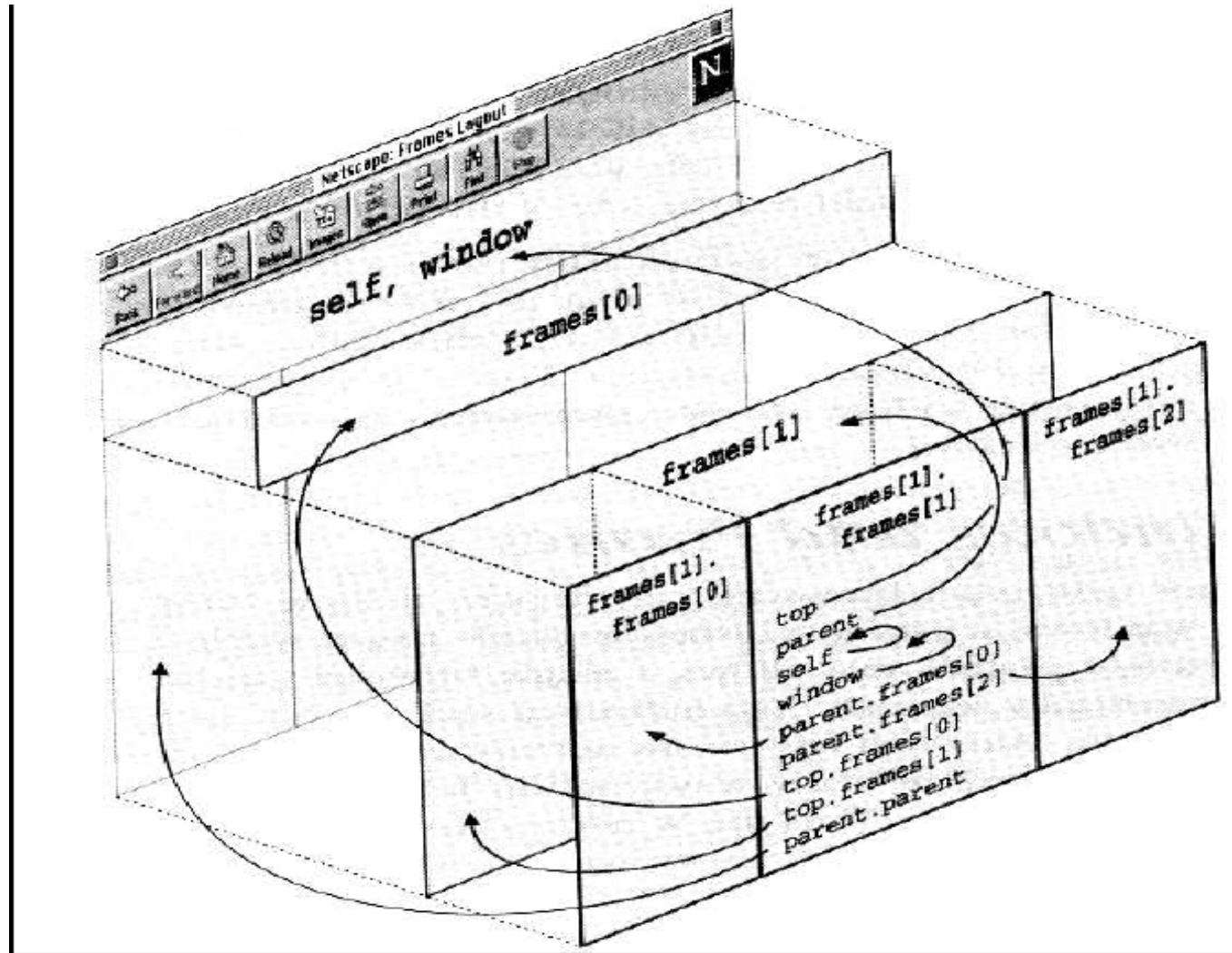
- Syntax:

`open ( <URL>,<windowName>[,<windowFeatures>] )`

- Examples:

```
window.open("http://www.polito.it","mywindow",  
"status=1,toolbar=1");  
window.open("http://www.javascriptcoder.com",  
"mywindow","menubar=1,resizable=1,width=350,  
height=250");  
mywindow.moveTo(0,0); // move new win to top left
```

# Window and Frame References



# Example

- In the Parent window:

...

```
<SCRIPT LANGUAGE="javascript" TYPE="text/javascript">
  leftPos = screen.width-225;
  car = "width=225,height=200,left="+leftPos+",top=0"
  newWindow = window.open("js_news.html", "newWin", car);
</SCRIPT>
<BODY>
<CENTER>
<H1>This is the main window</H1>
<H1>Here you will see online newspapers</H1>
</CENTER>
</BODY>
```

# Example

- In the New Window ([js\\_news.html](#)):

```
...
<SCRIPT LANGUAGE="javascript" TYPE="text/javascript">
    function updateParent(newURL)
    {opener.location.href = newURL;}
</SCRIPT>
<CENTER>
<H1>Control Panel</H1>
<H3>
<A HREF="javascript:updateParent('http://www.corriere.it')">
    Corriere della Sera</A><BR>
<A HREF="javascript:updateParent('http://www.lastampa.it')">
    La Stampa</A><BR>
<A HREF="javascript:updateParent('http://www.repubblica.it')">
    La Repubblica;</A>
</CENTER>
...
js-openwindow.html
```



# The navigator Object

- Gives access to browser features
- Main properties of `navigator`:
  - `appName` Browser code name
  - `appVersion` Browser application name
  - `cookieEnabled` Browser application version
  - `platform` true if cookies are enabled
  - `userAgent` The platform where the browser runs
  - `javaEnabled()` The user agent string the browser sends in the HTTP header
  - `javaEnabled()` true if java is enabled

# The screen Object

- Gives access to the features of the screen
- Main properties of **screen**:
  - **availHeight** height in pixels (excluding application bar)
  - **height** total height
  - **availWidth** width in pixels (excluding application bar)
  - **width** total width
  - **colorDepth** color depth in bits per pixel

# The location Object

- Indicates the URL displayed in a window or frame (in this case the **URL** attribute is generally preferred)
- The various parts of the URL (hostname, port, protocol,...) can be accessed separately as sub-properties
- The access is in **read/write** mode
- Writing a new value starts loading the new URL
- Main properties:
  - **href** complete URL
  - **host** hostname and port number
  - **protocol://hostname:port/pathname?search#hash**

# Example

`http://www.abc.com:555/catalog/search.php?query=JS  
&match=2#result`

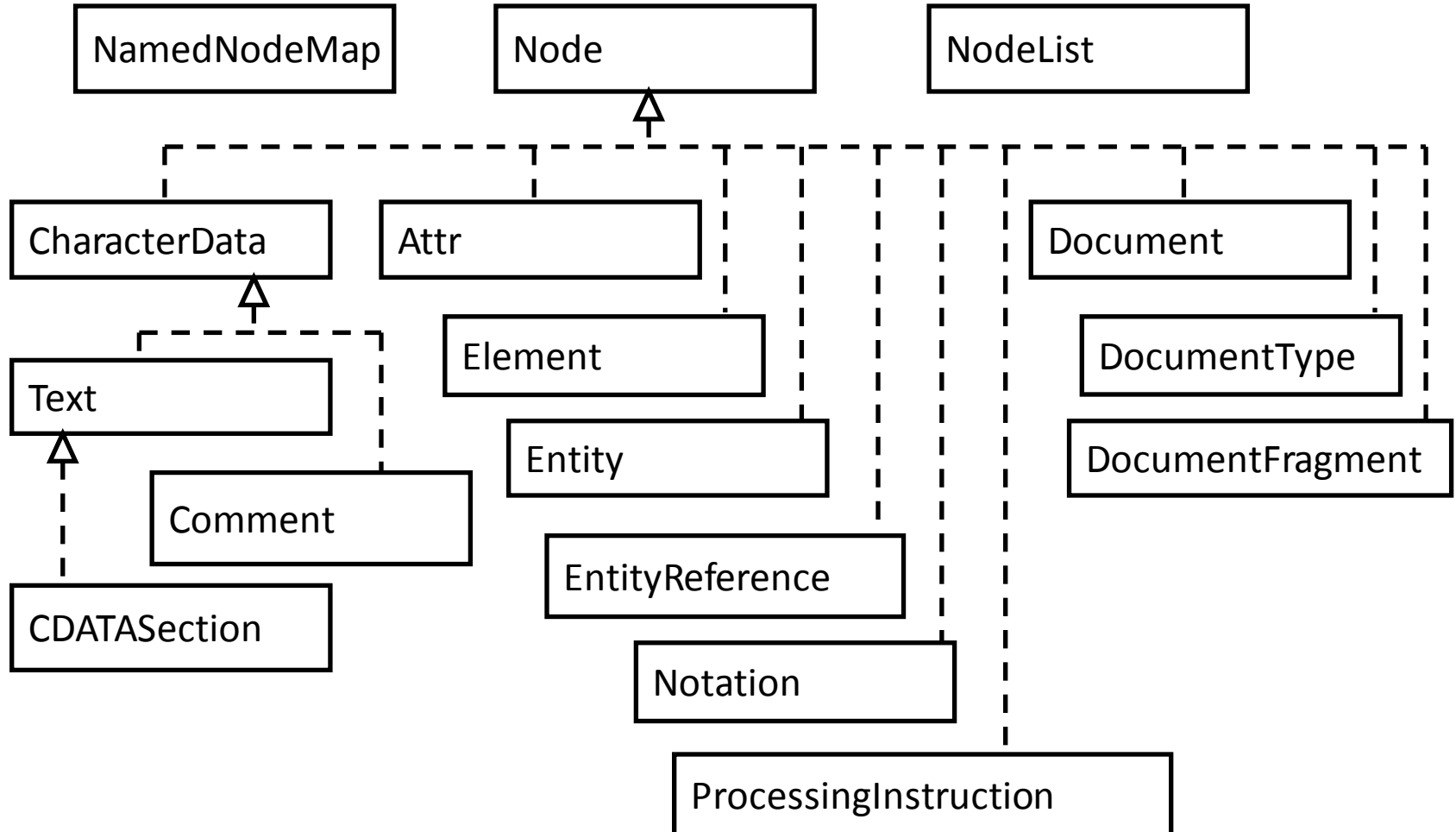
- `- protocol = "http: "`
- `- hostname = "www.abc.com"`
- `- port = "555"`
- `- host = "www.abc.com:555"`
- `- pathname = "/catalog/search.php"`
- `- search = "?query=JS&match=2"`
- `- hash = "#result"`

# The DOM API

- Standard cross-platform language-independent API for accessing the elements (and attributes) of a **document** (HTML, XML,...)
- DOM Functionalities are divided into **levels**:
  - **DOM 0** non-standard level including the basic functions that were provided by old browsers (kept only for compatibility)
  - **DOM 1** gives r/w access to all the elements of a document
  - **DOM 2** adds advanced access functions (including an event model), and access to CSS.
  - **DOM 3** adds more features, e.g. XPath support.
- Besides the standard functions, commercial browsers sometimes offer additional functions
  - Important: avoid them to get portability across browsers

W3C  
Std. {

# DOM Level 1 Main Classes



# The HTML-specific Document ( document Object )

- Main properties of **document**:
  - **title**                      the title of the document
  - **forms[ ]**                      the form elements in the document
  - **anchors[ ]**                      the `<a>` elements in the document
  - **links[ ]**                      the `<link>` elements in the document
  - **images[ ]**                      the image elements in the document
  - **body**                      the document body element
- Main methods of **document**:
  - **write()**                      writes to the current document position (where rendering has arrived)
  - **writeln()**                      like write but adds a newline at the end

# Accessing Document Elements

- The access path may be very long. For example:
  - `parent.frames[0].document.forms[0].elements[3].button[2].value`
- Access to elements can be simplified by giving **id** or **name** attributes to the elements
- Of course id or name must be **unambiguous**



# Example

```
<html>
<head>
<script type="text/javascript">
i=1;
function change (){
    if (i==0) {document.images["banner"].src="questions.jpg"; i=1}
    else      {document.images["banner"].src="caravaggio.jpg"; i=0}
}
</script>
</head>
<body>

<input type="button" value="Change" onclick="change()">
</body>
</html>
```

[js-swap.html](#)

# Other Examples

```
<FORM NAME="myform">
<INPUT TYPE="TEXT" NAME="text">
</FORM>
...
document.forms["myform"].elements["text"].value = "New Text";
...
```

```
<FORM>
<INPUT TYPE="TEXT" id="text">
</FORM>
...
document.getElementById("text").value = "New Text";
...
```

# Reacting to Events

- The browser can detect the occurrence of certain events
- A function that reacts to the event (**handler**) can be associated with each kind of event

# Event Kinds

- Abort
- Blur
- Change
- Click
- DblClick
- Error
- Focus
- KeyDown
- KeyPress
- KeyUp
- Load
- MouseDown
- MouseMove
- MouseOut
- MouseOver
- MouseUp
- Move
- Reset
- Resize
- Select
- Submit
- Unload

# Specifying Event Handlers

- An event handler can be associated with an **event kind** and with an **HTML element** by an attribute named after the event kind
- Syntax:

***<<TagName> on<Event>="<function>"***

- Example:

```
<SCRIPT TYPE="text/javascript">
  function hello() { alert("You entered the image"); }
  function bye() { alert("You left the image"); }
</SCRIPT>
<IMG SRC="questions.jpg" border="0"
      onMouseOver="hello()"
      onMouseOut="bye()">
```

[js-mouse.html](http://www.w3schools.com/js/js-mouse.html)

# Other Example

```
<head>
<script type="text/javascript">
var w;
function myOpenWindow() {
    w=window.open('', '', 'width=100,height=100');
    w.focus();
}
function myResize() {
    w.resizeTo(500,500);
    w.focus();
}
</script>
</head>
<body>
<button onclick="myOpenWindow()">Create window</button>
<button onclick="myResize()">Resize window</button>
</body>
```

<js-resize.html>

# Timeout

- The timing of script actions can be controlled by using timeouts.
- The methods that control timeouts are:
  - **setTimeout**( *<statement>*, *<ms>*)
    - Starts a timer and sets the timeout after *<ms>* milliseconds
    - Returns a reference to the timeout
    - On timeout expiration *<statement>* will be executed
  - **clearTimeout**( *<timeoutId>*)
    - Cancels the timeout referenced by *<timeoutId>* (if not yet expired)

# Timeout Example

- HTML:

```
<IMG name= "image" src="questions.jpg"
  height=200 width=200 >
```

```
<FORM name="myform">
```

```
In how many seconds do you want to change image?
&nbsp;
```

```
<INPUT TYPE="text" name="seconds" size="2">
&nbsp;
```

```
<INPUT TYPE="button" value="Change"
  onClick="setTimer()"> &nbsp; &nbsp;
```

```
<INPUT TYPE="button" value="Cancel"
  onClick="cancelTimer()">
```

```
</FORM>
```



# Timeout Example

- JavaScript:

```
var tID = 0;
function change() {
    document.image.src="Caravaggio.jpg";
};
function setTimer() {
    var sec =
        parseInt(document.myform.seconds.value);
    if(!tID)
        tID= setTimeout("change()",sec*1000);
}
function cancelTimer() {
    if (tID) {
        clearTimeout(tID);  tID=0;
    }
};
```

[js-timeout.html](#)

# Exercise

- Modify the script, so that it changes two given images alternatively

# Other Example

```
<input type="button" name="clickMe"  
      value="Click and wait!"  
      onclick="setTimeout('alert(`Hello!`)',  
                           5000)"  
>
```

# Periodic Execution

- The periodic execution of a script can be programmed by the methods:
  - **setInterval**( *<statement>*, *<ms>*)
    - Causes the execution of *<statement>* every *<ms>* milliseconds
    - Returns a reference to the programmed interval
  - **setInterval**( *<function>*, *<ms>*, *<par<sub>1</sub>>* ,... , *<par<sub>n</sub>>*)
    - Causes the execution of *<function>* with parameters *<par<sub>1</sub>>* ... *<par<sub>n</sub>>* every *<ms>* milliseconds
    - Returns a reference to the programmed interval
  - **clearInterval**( *<Intervald>*)
    - Cancels the action of the interval referenced by *<Intervald>*

# Example (1)

- **HTML:**

```
<TABLE> <TR>
<TD> <IMG SRC="a.gif" height="200"
NAME="image" border="0">
<TD>
<FORM>
<INPUT TYPE="button" VALUE="Start"
onClick="start()"><P>
<INPUT TYPE="button" VALUE="Stop"
onClick="stop()">
</FORM>
</TABLE>
```

# Example (2)

- **JavaScript:**

```
<SCRIPT TYPE="text/javascript">
var intID = 0; var variable = 0;
var images = new Array();
var counter=0;
for(i=0;i<4;i++)images[i] =
    new Image();
images[0].src = "questions.jpg";
images[1].src = "Caravaggio.jpg";
images[2].src = "pyramids.jpg";
images[3].src = "Renoir.jpg";
</SCRIPT>
```

# Example (3)

## JavaScript Functions:

```
function start() {  
    intID = setInterval("change()",1000);  
}  
  
function change() {  
    counter = (++counter)%4;  
    document.image.src = images[counter].src;  
}  
  
function stop() {  
    clearInterval(intID);  
}
```

[js-interval.html](#)

# More about Error Handling

- The event handler `window.onerror` is an alternative to try-catch for error handling
- Not supported by all browsers
- The handler function receives 3 parameters from the browser
  - Error type (string)
  - The document where the error occurred (url)
  - The line number where the error occurred



# Example

```
window.onerror = handleError;  
function handleError(error, url, line) {  
    var msg = ""  
    msg = "Error type: " + error + "\n";  
    msg = msg + "Document: " + url + "\n";  
    msg = msg + "Line: " + line;  
    alert(msg);  
}
```

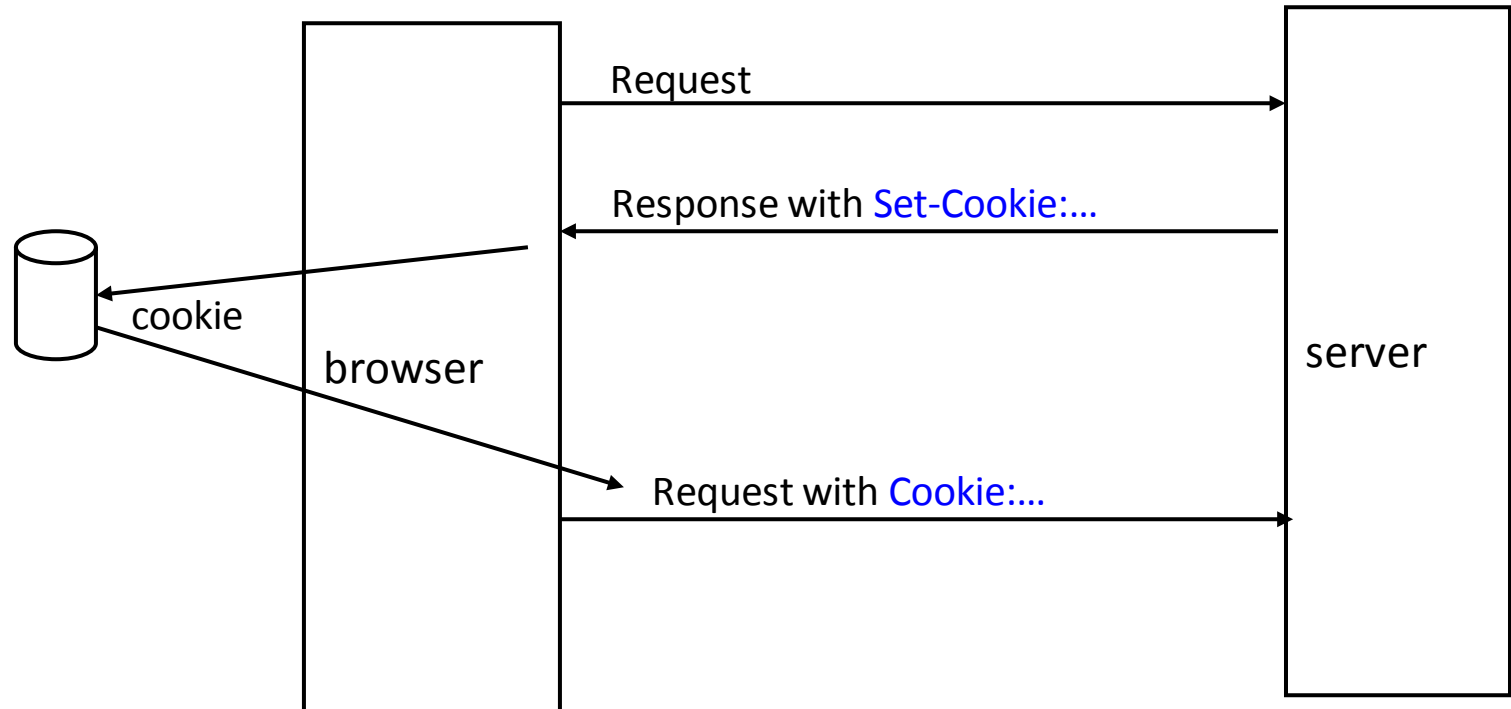
# Cookies

- Cookies are short textual data packets that a browser can store persistently
- Generally cookies are generated by the server and sent to the browser (using the **set-cookie** header) but client-side scripts can set cookies by themselves
- A browser can send back a cookie to a server (and even change it before sending).
- Typical uses of cookies:
  - User identification,
  - User profiling
  - Storage of user data (e.g. preferences) without server-side management

# Cookie Standardization


- Initially introduced by Netscape, the cookies mechanism has been later standardized as RFC:
  - Rfc 2965: HTTP State Management Mechanism
  - Rfc 2964: Use of HTTP State Management

# How HTTP Cookies Work



# Set-Cookie Syntax

Set-Cookie: `<name>=<value>;` *<optionalArgs>*  
*expires=<date>; path=<path>; domain=<domain>; secure*



- `<name>`, `<value>`: **cookie name and value**
- *<date>*: **expiry date**. If not specified, the cookie is deleted when the browser session terminates
- *<domain>*, *<path>*: the lists of **domains and paths** the cookie will have to be sent to (if not specified, they are the ones of the current request)
- **secure**: if present requires that cookies are sent only on secure (HTTPS) channels

# What Cookies Should be Sent?

- Browsers normally send cookies automatically (users may disable cookies)
- At each request, the browser selects the cookies to send by looking at their DOMAIN and PATH attributes:
  - A cookie can be sent only if its DOMAIN matches the *final* part of the destination hostname

**Example:** if DOMAIN="polito.it", the cookie can be sent to hosts www.polito.it, www.webservices.polito.it, etc.
  - A cookie can be sent only if its PATH attribute matches the *initial* part of the destination path

**Example:** if PATH="/foo", the cookie can be sent when the destination paths are /foo/, /foo/bar.html, etc.

# Sending Cookies

- Cookies are sent by the HTTP header

`Cookie: <name1>=<value1>; <name2>=<value2>; ...`

- Cookie names and values cannot include **space** , ; “ \
- In Javascript
  - A cookie can be **added/modified or read** by the method `document.cookie`
  - Strings used as cookie values should be encoded with `escape()` and decoded with `unescape()`

# Example: Reading Cookies

```
var cook = document.cookie; /* reads all cookies */
var i=cook.indexOf("par="); /* search for cookie that
                             matches "par=" */

if (i!=-1) { // found
    var k=i+4; // start of value
    var end=cook.indexOf(";",k);
    if(end=-1) end=cook.length;
    var value=cook.substring(k,end);
    value=unescape(value);

    ...
}
```



# Choosing Cookie Names

- The name can be used for storing information about the meaning of a cookie or any other information associated with the cookie
- The format is free

# Example: Writing Cookies

```
var cook;  
cook="Date"+ escape(document.lastModified);  
var expiry=new Date();  
// set expiry in 1 year  
expiry.setFullYear= expiry.getFullYear+1;  
// convert to GMT date  
expire="expires="+expiry.toGMTString()+";";  
// write cookie  
document.cookie=cook+";"+expire;
```

# Example: Deleting Cookies

- A cookie can be cancelled by simply setting its expiration date to a past date
- Example:

```
var cook=document.cookie;
var i=cook.indexOf("expires=");
if(i!=-1) {
    i+=8;//advance after expires=
    var k=cook.indexOf(";",i);
    if (k=-1) k=cook.length;
    var expiration=new Date(1930,12,1);
    substr=cook.slice(i,k);
    cook=cook.replace(substr,
        escape(expiration.toGMTString()));
}
```

# Javascript and CSS

- DOM 2 includes functions for CSS handling
- Not fully supported by browsers
- CSS handling functions operate on element objects
- Element objects can be found by their id:

```
var elem=getElementById("foto1");
```

# Changing CSS Properties

- Change visibility

```
var elem=getElementById("text1");  
elem.style.visibility="hidden";  
elem.style.visibility="visible";
```

- Style properties have same names as CSS properties
- Examples:

```
elem.style.top="223px";  
elem.style.color="darkred";  
elem.style.fontSize="12px"
```

# Simple Animations

- Can be implemented setting `interval` calls
  - Style properties of some elements are changed periodically
- Example:

[js-animation.html](#)

# Example: Style

```
<style name="first" type="text/css">
#banner
{
  left:0px;top:0px;font-size:x-medium;
  font-family:
    Roman;color:#FF00FF;width:300px;position:relative;
}
</style>
```

# Example: HTML

```
<body>
<div id="banner" > Hello!</div>
<br><br> <br>

<br>
<form name=myform>
<input type=button onclick=startBanner() value="Start banner">
<input type=button onclick=stopBanner() value="Stop banner">
<br>
<input      type=button      onclick=startBlinking()      value="Start
    blinking">
<input type=button onclick=stopBlinking() value="Stop blinking">
</body>
```



# Javascript (1)

```
<script type="text/javascript">
var xmin=0; var x=0;
var xmax=300;
var deltax=5;
var intBlink=0;
var intBanner=0;
function move() {
    var elem=document.getElementById("banner");
    if (x>=xmax) x=0
    else x+=10;

    elem.style.left=x+"px";
}
```

# Javascript (2)

```
function blink() {  
    var elem=document.getElementById("logo");  
    if(elem.style.visibility=="hidden")  
        elem.style.visibility="visible";  
    else  
        elem.style.visibility="hidden";  
};
```

```
function startBanner() {  
    if (!intBanner)  
        intBanner=setInterval(move,40);  
};
```

```
function stopBanner() {  
    clearInterval(intBanner);  
    intBanner=0;  
};
```

# Javascript (3)

```
function startBlinking() {  
    if (!intBlink)  
        intBlink=setInterval(blink,500);  
};
```

```
function stopBlinking() {  
    clearInterval(intBlink);  
    intBlink=0;  
};  
</script>
```

# Other Style Attributes

- If `elem` is an HTML element

<code>elem.style.left</code>	top left position
<code>elem.style.top</code>	top position
<code>elem.style.zIndex</code>	depth
<code>elem.style.background</code>	background color
<code>elem.style.color</code>	element color

- If `window` is the current window

<code>window.innerWidth</code>	window inner width
<code>window.innerHeight</code>	window inner height

# Other Graphic Effects

- Acting on style elements, timers and intervals it is possible to build several graphic elements, such as
  - Pull down menus
  - Callouts (page elements that appear when passing over other elements)
  - ...