JavaScript: fundamentals, concepts, object model

Prof. Ing. Andrea Omicini
II Facoltà di Ingegneria, Cesena
Alma Mater Studiorum, Università di Bologna
andrea.omicini@unibo.it

JavaScript

- A scripting language: interpreted, not compiled
- History:
  - Originally defined by Netscape (LiveScript) – Name modified in JavaScript after an agreement with Sun in 1995
  - Microsoft calls it JScript (minimal differences)
  - Reference: standard ECMAScript 262
- Object based (but not object oriented)
- JavaScript programs are directly inserted in the HTML source of web pages

The Web Page

```
<html>
  <head><title>...</title></head>
  <body>
  ...
  <script language="JavaScript">
  <!-- HTML comment to avoid puzzling old browsers
  ... put here your JavaScript program ...
  // JavaScript comment to avoid puzzling old browsers -->
  </script>
  </body>
</html>
```

An HTML page may contain multiple `<script>` tags

Document Object Model

- JavaScript as a language references the Document Object Model (DOM)
- Following that model, every document has the following structure:

```
window
    document
    ...
```

- The `window` object represents the current object (i.e. `this`) the current browser window
- The `document` object represents the content of the web page in the current browser window

The `document` object

- The document object represents the current web page (not the current browser window!)
- You can invoke many different methods on it. The write method prints a value on the page:

```
document.write("Scrooge McDuck")
document.write(18.45 - 34.44)
document.write('Donald Duck')
document.write('<IMG src="image.gif">')
```

- The `this` reference to the `window` object is omitted: `document.write` is equivalent to `this.document.write`

The `window` object (1/2)

- The `window` object is the root of the DOM hierarchy and represents the browser window
- Amongst the `window` object's methods there is `alert`, which makes an alert window displaying the given message appear

```
x = -1.55; y = 31.85; sum = x + y
message = "Somma di " + x + " e " + y
alert(message + ": " + sum) // returns undefined
```

- You can use `alert` in an HTML anchor

The `window` object (2/2)

Other methods of the `window` object:

- use `confirm` to display a dialog to confirm or dismiss a message
  - returns a boolean value: `false` if the Cancel button has been pushed, `true` if the OK button has been pushed
- use `prompt` to display a dialog to input a value
  - returns a `string` value containing the input

The DOM model

The `window` object's main components:

- `self`
- `window`
- `parent`
- `top`
- `navigator`
  - `plugins` (array), `navigator`, `mimeTypes` (array)
- `frames` (array)
- `location`
- `history`
- `document`

...and here follows an entire hierarchy of objects

The `document` object

The `document` object's main components (all arrays):

- `forms`
- `anchors`
- `links`
- `images`
- `applets`

The `document` object's main API methods:

- `getElementsByTagName(tagname)`
- `getElementById(elementId)`
- `getElementsByName(elementName)`

Referencing an element in a document

- An element in a document is referred to by the value of its `id` attribute (or the `name` attribute in older browsers)
  - e.g. for an image identified as `image0` you would call `document.getElementById("image0")`
  - or use the document properties through an array: `document.images["image0"]`
  - then, to modify e.g. that image's width, you would write `document.images["image0"].width = 40`

Strings

- Strings can be delimited by using single or double quotes
- If you need to nest different kind of quotes, you have to alternate them
  - e.g. `document.write('<img src="img.gif">')`
  - e.g. `document.write("<img src='img.gif'>")`
- Use + to concatenate strings
  - e.g. `document.write('donald' + 'duck')`
- Strings are JavaScript objects with properties, e.g. `length`, and methods, e.g. `substring(first, last)`

Constants and comments

- Numeric constants are sequences of numeric characters not enclosed between quotes - their type is `number`
- Boolean constants are `true` and `false` - their type is `boolean`
- Other constants are `null`, `NaN`, `undefined`
- Comments can be
  - `// on a single line`
  - `/* multi line */`

Expressions

These are legal expressions in JavaScript

- numeric expressions, with operators like + - * / % …
- conditional expressions, using the ?: ternary operator
- string expressions, concatenating with the + operator
- assignment expressions, using =

Some examples

- `document.write(18/4)`
- `document.write(3>5 ? 'yes' : "no")`
- `document.write("donald" + 'duck')`

Variables

- Variables in JavaScript are dynamically typed: you can assign values of different types to the same variable at different times

```
a=19; b='bye'; a='world'; // different types!
```

- Legal operators include increment (++), decrement (--), extended assignment (e.g. +=)

Variables and scope

- Variable scope in JavaScript is
  - global for variables defined outside functions
  - local for variables explicitly defined inside functions (received parameters included)
- Warning: a block does not define a scope

```
x = '3' + 2 // the string '32'
{
   { x = 5 } // internal block
   y = x + 3 // here x is 5, not '32'
}
```

Dynamic types

- The `typeof` operator is used to retrieve the (dynamic) type of an expression or a variable

```
typeof(18/4) returns number
typeof "aaa" returns string
typeof false returns boolean
typeof document returns object
typeof document.write returns function
```

- When used with variables, the value returned by typeof is the current type of the variable

```
a = 18; typeof a // returns number
a = 'hi'; typeof a // returns string
```

Instructions

- Instructions must be separated by an end-of-line character or by a semicolon

```
alpha = 19 // end-of-line
bravo = 'donald duck'; charlie = true
document.write(bravo + alpha)
```

- Concatenation between strings and numbers leads to an automatic conversion of the number value into a string value (be careful…)

```
document.write(bravo + alpha + 2)
document.write(bravo + (alpha + 2))
```

Control structures

- JavaScript features the usual control structures: `if`, `switch`, `for`, `while`, `do/while`
- Boolean conditions in an `if` can be expressed using the usual comparison operators (==, !=, >, <, >=, <=) and logic operators (&&, ||, !)
- Besides there are special structures used to work on objects: `for/in` and `with`

Functions definition

- Functions are introduced by the keyword `function` and their body is enclosed in a block
- They can be either procedures or proper functions (there's no keyword `void`)
- Formal parameters are written without their type declaration
- Functions can be defined inside other functions

```
function sum(a,b) { return a+b }

function printSum(a,b) {
   document.write(a+b)
}
```

Function parameters

- Functions are called in the usual way, giving the list of actual parameters
- The number of actual parameters can be different from the number of formal ones
- If actual parameters are more than necessary, extra parameters are ignored
- If actual parameters are less than necessary, missing parameters are initialized to `undefined`
- Parameters are always passed by value (working with objects, references are copied)

Variable declarations

- Variable declarations can be explicit or implicit for global variables, but must necessarily be explicit for local variables
- A variable is explicitly declared using `var`

```
var goofy = 19 // explicit declaration
pluto = 18 // implicit declaration
```

- Implicit declaration always introduces global variables, while explicit declaration has a different effect depending on the context where it is located

Explicit variable declarations

- Outside functions, the `var` keyword is not important: the variable is defined as global
- Inside functions, using `var` means to introduce a new local variable having the function as its scope
- Inside functions, declaring a variable without using `var` means to introduce a global variable

```
x = 6 // global
function test() {
  x = 18 // global
}
test()
// the value of x is 18
```

```
var x = 6 // global
function test() {
  var x = 18 // local
}
test()
// the value of x is 6
```

Referencing environment

- Using an already declared variable, its name resolution starts from the environment local to its use
- If the variable is not defined in the environment local to its use, the global environment is checked for name resolution

```
f = 3
function test() {
  var f = 4
  g = f * 3
}
test(); g // 12
```

```
f = 3
function test() {
  var g = 4
  g = f * 3
}
test(); g // nd
```

```
f = 3
function test() {
  var h = 4
  g = f * 3
}
test(); g // 9
```

Functions and closures (1/3)

- Since JavaScript is an interpreted language and given the existence of a global environment...
- When a function uses a symbol not defined inside its body, which definition holds for that?
  - Does the symbol use the value it holds in the environment where the function is defined, or...
  - does the symbol use the value it holds in the environment where the function is called?

Functions and closures (2/3)

```
var x = 20
function testEnv(z) { return z + x }
alert(testEnv(18)) // definitely displays 38
function newTestEnv() {
   var x = -1
   return testEnv(18) // what does it return?
}
```

- The `newTestEnv` function redefines `x`, then invokes `testEnv`, which uses `x`... but, which `x`?
- In the environment where `testEnv` is defined, the symbol `x` has a different value from the environment where `testEnv` is called

Functions and closures (3/3)

```
var x = 20
function testEnv(z) { return z + x }
function newTestEnv() {
   var x = -1
   return testEnv(18) // what does it return?
}
```

- If the calling environment is used to resolve symbols, a dynamic closure is applied
- If the defining environment is used to resolve symbols, a lexical closure is applied
- JavaScript uses lexical closures, so `newTestEnv` returns 38, not 17

Functions as data

- Variables can reference functions

```
var square = function(x) { return x*x }
```

- Function literals have not a name: they are usually invoked by the name of the variable referencing them

```
var result = square(4)
```

- Assignments like `g = f` produce aliasing
- This enables programmers to pass functions as parameters to other functions

```
function exe(f, x) { return f(x) }
```

Functions as data - Examples

Given `function exe(f, x) { return f(x) }`

`exe(Math.sin, .8)` returns `0.7173560908995228`

`exe(Math.log, .8)` returns `-0.2231435513142097`

`exe(x*x, .8)` throws an error because `x*x` is not a function object in the program

`exe(fun, .8)` works only if the `fun` variable references a function object in the program

`exe("Math.sin", .8)` throws an error because a string is passed, not a function: don't mistake a function for its name

Functions as data - Consequences

- You need to have a `function` object (not just its name) to use a function
- You cannot use functions as data to execute a function knowing only its name or its code

```
exe("Math.sin", .8) // error
exe(x*x, .8) // error
```

- How to solve this problem?
  - Access to the function using the properties of the global object
  - Build an appropriate `function` object

Objects

- An object is a data collection with a name: each datum is called property
- Use the dot notation to access any property, e.g. `object.property`
- A special function called constructor builds an object, creating its structure and setting up its properties
- Constructors are invoked using the `new` operator
- There are no classes in JavaScript: the name of the constructor can be choosed by the user

Defining objects

- The structure of an object is defined by the constructor used to create it
- Initial properties of the object are specified inside the constructor, using the dot notation and the `this` keyword
- The `this` keyword is necessary, otherwise properties would be referenced by the environment local to the constructor function

```
Point = function(i, j) {
  this.x = i
  this.y = j
}
```

```
function Point(i, j) {
  this.x = i
  this.y = j
}
```

Building objects

- To build an object, apply the `new` operator to a constructor function

```
p1 = new Point(3, 4)
p2 = new Point(0, 1)
```

- The argument of new is just a function name, not the name of a class
- Starting with JavaScript 1.2 just listing couples of properties and values between braces

```
p3 = {x:10, y:7}
```

Accessing object properties

- All properties of an object are public

```
p1.x = 10 // p1 passes from (3,4) to (10,4)
```

- There are indeed some invisible system properties you can not enumerate using the usual appropriate constructs
- The `with` construct let you access several properties of an object without repeating its name every time

```
with (p1) x = 22, y = 2
with (p1) {x = 3; y = 4}
```

Adding and removing properties

- Constructors only specify initial properties for an object: you can dynamically add new properties by naming them and using them

```
p1.z = -3 // from {x:10, y:4} to {x:10, y:4,
z:-3}
```

- It is possible to dynamically remove properties using the delete operator

```
delete p1.x // from {x:10, y:4, z:-3} to {y:4,
z:-3}
```

Methods for (single) objects

- Methods definition is a special case of property addition where the property is a function object

```
p1.getX = function() { return this.x }
```

- In this case, a method is defined for a single object, not for every instance created using the `Point` constructor function

Methods for multiple objects

- You can define the same method for multiple objects by assigning it to other objects

```
p2.getX = p1.getX
```

- To use the new method on the `p2` object, just call it using the `()` invoke operator

```
document.write(p2.getX() + "<br/>")
```

- If a nonexistent method is invoked, JavaScript throws a runtime error and halts execution

Methods for objects of a kind

- Since the concept of class is missing, ensuring that objects "of the same kind" have the same behaviour requires an adequate methodology
- A first approach is to define common methods in the constructor function

```
Point = function(i, j) {
   this.x = i; this.y = j
   this.getX = function() { return x }
   this.getY = function() { return y }
}
```

- Another approach is based on the concept of prototype (see later)

Simulating private properties

- Even if an object's properties are public, it is possible to simulate private properties using variables local to the constructor function

```
Rectangle = function() {
   var sideX, sideY
   this.setX = function(a) { sideX = a }
   this.setY = function(a) { sideY = a }
   this.getX = function() { return sideX }
   this.getY = function() { return sideY }
}
```

- While the four methods are publicly visible, the two variables are visible in the constructor's local environment only, being matter-of-factly private

Class variables and methods

- Class variables and methods can be modeled as properties of the constructor function object

```
p1 = new Point(3, 4); Point.color = "black"
Point.commonMethod = function(...) { ... }
```

- The complete `Point.property` notation is necessary even if the property is defined inside the constructor function, because property alone would define a local variable to the function, not a property of the constructor

`Function` objects (1/2)

- Every function is an object built on the basis of the `Function` constructor
  - implicitly, building functions inside the program by using the `function` construct
    - its arguments are the formal parameters of the function
    - the body (the code) of the function in enclosed in a block
  - e.g. `square = function(x) { return x*x }`
  - the construct is evaluated only once, it's efficient but not flexible

`Function` objects (2/2)

- Every function is an object built on the basis of the `Function` constructor
  - explicitly, building functions from strings by using the `Function` constructor
    - its arguments are all strings
    - first N-1 arguments are the names of the parameters of the function
    - the last argument is the body (the code)
  - e.g. `square = new Function('x', 'return x*x')`
  - the construct is evaluated every time it's read, it's not efficient but very flexible

Functions as data - Revision (1/4)

- The `exe` function executes a function

```
function exe(f, x) { return f(x) }
```

- It works only if the `f` argument represents a `function` object, not a body code or a string name

```
exe(x*x, .8) // error
exe("Math.sin", .8) // error
```

- These cases become manageable by using the `Function` constructor to dynamically build a function object

Functions as data - Revision (2/4)

- Dynamic building using the `Function` constructor
  - when only the body is known

```
exe(x*x, .8) // error
exe(new Function('x', 'return x*x'), .8) //
returns .64
```

  - when only the name is known

```
exe('Math.sin', .8) // error
exe(new Function('z', 'return Math.sin(z)'), .8) //
returns 0.7173560908995228
```

Functions as data - Revision (3/4)

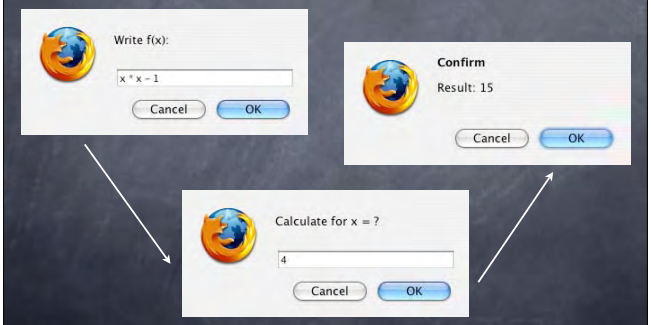
- Generalizing the approach:

```
var fun = prompt('Write f(x): ')
var x = prompt('Calculate for x = ?')
var f = new Function('x', 'return ' + fun)
```

- The user can now type the code of the desired function and the value where to calculate it, then invoke it using a reflexive mechanism
- Show the result using

```
confirm('Result: ' + f(x))
```

Functions as data - Revision (4/4)



Functions as data - A problem

- Values returned by prompt are strings: so the `+` operation is interpreted as a concatenation of strings rather than a sum between numbers
- If the user gives `x+1` as a function, when `x=4` the function returns `41` as a result
- Possible solutions:
  - let the user write in input an explicit type conversion, e.g. `parseInt(x) + 1`
  - impose the type conversion from within the program, e.g. `var x = parseInt(prompt(...))`

`Function` objects - Properties

Static properties (available while not executing):

- `length` - the number of formal expected parameters

Dynamic properties (available during execution only):

- `arguments` - array containing actual parameters
- `arguments.length` - number of actual parameters
- `arguments.callee` - the executing function itself
- `caller` - the caller (`null` if invoked from top level)
- `constructor` - reference to the constructor object
- `prototype` - reference to the prototype object

`Function` objects - Methods

Callable methods on a `function` object:

- `toString` - returns a string representation of the function
- `valueOf` - returns the function itself
- `call` and `apply` - call the function on the object passed as a parameter giving the function the specified parameters
  - e.g. `f.apply(obj, arrayOfParameters)` is equivalent to `obj.f(arrayOfParameters)`
  - e.g. `f.call(obj, arg1, arg2, ...)` is equivalent to `obj.f(arg1, arg2, ...)`

`call` and `apply` - Example 1

- Definition of a function object

```
test = function(x, y, z) { return x + y + z }
```

- Invocation in the current context

```
test.apply(obj, [3, 4, 5])
test.call(obj, 8, 1, -2)
```

- Parameters to the callee are optional
- In this example the receiving object `obj` is irrelevant because the invoked `test` function does not use `this` references in its body

`call` and `apply` - Example 2

- A function object using this references

```
test = function(v) { return v + this.x }
```

- In this example the receiving object is relevant because it determines the evaluation environment for the variable `x`

```
x = 88
test.call(this, 3)

// Result: 3 + 88 = 91
```

```
x = 88
function Obj(u) {
  this.x = u
}
obj = new Obj(-4)
test.call(obj, 3)

// Result: 3 + -4 = -1
```