JavaScript: Fundamentals, Concepts, Object Model

Prof. Ing. Andrea Omicini
Ingegneria Due, Università di Bologna a Cesena
andrea.omicini@unibo.it
2007-2008
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 ECMA-Script 262

- Object based (but not object oriented)
- JavaScript programs are directly inserted in the HTML source of web pages
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:
  
  ```javascript
  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 visualising entity.
- The visualised entity.
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:

```javascript
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:

```javascript
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 appear, displaying the given message:

```javascript
x = -1.55; y = 31.85; sum = x + y;
mess = "La somma di " + x + " e " + y + " è " + sum;
alert(mess); // returns undefined
```
- You can use alert in an HTML anchor.
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 / modifying 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 – deprecated!

- 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:

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

- Variables in JavaScript are dynamically typed: you can assign values of different types to the same variable at different times
  
  ```javascript
  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
  ```javascript
  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
window.alert(bravo + alpha)

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

window.alert(bravo + alpha + 2)
window.alert(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.

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

function printSum(a, b) {
    window.alert(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 can be explicit or implicit for global variables, but must necessarily be explicit for local variables.

A variable is explicitly declared using `var`:

```javascript
var goofy = 19 // explicit declaration
pluto = 18 // explicit 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.

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

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

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

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

```javascript
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)

```javascript
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)

```javascript
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
  ```javascript
  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
  ```javascript
  var result = square(4)
  ```
- Assignments like `g = f` produce aliasing
- This enables programmers to pass functions as parameters to other functions
  ```javascript
  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 an expression, 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.
  ```javascript
  exe("Math.sin", .8) // error
  exe(x*x, .8) // error
  ```

- How to solve this problem?
  - Access 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.

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

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

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

\[
p_1 = \text{new Point}(3, 4)\\
p_2 = \text{new Point}(0, 1)
\]

The argument of new is just a function name, not the name of a class.

Starting with JavaScript 1.2 objects can be built just listing couples of properties and values between braces:

\[
p_3 = \{x:10, y:7\}
\]
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

\[
\text{p1.z} = -3 \\
\text{// from } \{x:10, \ y:4\} \ \text{to } \{x:10, \ y:4, \ z:-3\}
\]

It is possible to dynamically remove properties using the delete operator

\[
\text{delete \ p1.x} \\
\text{// from } \{x:10, \ y:4, \ z:-3\} \ \text{to } \{y:4, \ z:-3\}
\]
Methods for (single) objects

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

```javascript
pl.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

\[ p2 \text{.getX} = p1 \text{.getX} \]

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

```javascript
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

```javascript
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.

```javascript
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:

```javascript
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 is enclosed in a block.

Example: `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
The `exe` function executes a function

```javascript
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

```javascript
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
    ```javascript
    exe(x*x, .8) // error
    exe(new Function('x', 'return x*x'), .8) // returns .64
    ```
  - when only the name is known
    ```javascript
    exe('Math.sin', .8) // error
    exe(new Function('z', 'return Math.sin(z)'), .8) // returns 0.7173560908995228
    ```
Generalizing the approach:

```javascript
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)

Write $f(x)$:

$\ x \times x - 1$

Confirm

Result: 15

Calculate for $x = ?$

4

OK
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
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

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

Invocation in the current context

```javascript
    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

```javascript
    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`

```javascript
    x = 88

    test.call(this, 3)

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

```javascript
    x = 88

    function Obj(u) {
        this.x = u
    }

    obj = new Obj(-4)
    test.call(obj, 3)

    // Result: 3 + -4 = -1
```
Prototypes (1/2)

- Every object has always a prototype specifying its basic properties
- The prototype itself is an object
- If P is prototype of X, every property of P is also available as a property of X and thus redefinable by X
- The prototype is stored in a typically invisible system property called __proto__
Prototypes (2/2)

- Every constructor has a building prototype defined in its `prototype` property.
- It serves to define the properties of the objects it builds.
- By default, the building prototype coincides with the prototype, but while the latter is unchangeable, the former can be modified.
- The modifiability of the building prototype leads to prototype-based inheritance techniques.
Prototypes: architecture

**Object**

<table>
<thead>
<tr>
<th><strong>proto</strong></th>
<th>prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>specific properties for the object</td>
<td></td>
</tr>
</tbody>
</table>

**Constructor**

<table>
<thead>
<tr>
<th><strong>proto</strong></th>
<th>prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>prototype</td>
<td>building prototype (by default it is the same as the prototype)</td>
</tr>
</tbody>
</table>
Predefined prototypes

JavaScript makes available a series of predefined constructors whose prototype is the prototype for all the objects of that kind.

- The prototype of the `Function` constructor is the prototype for every function.
- The prototype of the `Array` constructor is the prototype of all the arrays.
- The prototype of the `Object` constructor is the prototype of all user defined objects built using the `new` operator.
- Other predefined constructors are `Number`, `Boolean`, `Date`, and `RegExp`.
Since constructors themselves are objects, they have a prototype too.

A taxonomy of prototypes is created, rooted in the prototype for the `Object` constructor.

The prototype of `Object` defines the properties:

- `constructor` - the function which built the object
- `toString()` - a method to print the object
- `valueOf()` - returns the underlying primitive type

These properties are available for every object (functions and constructors included).
All functions and in particular all constructors are attached to the prototype of `Function`.

That prototype defines common properties (e.g. arguments) for every function (including constructors) and inherits properties from the prototype of `Object` (e.g. constructor).
Experiments

The predefined method `isPrototypeOf()` tests if an object is included in another object's chain of prototypes

```javascript
Object.prototype.isPrototypeOf(Function) // true
Object.prototype.isPrototypeOf(Array) // true
```

The `Point` constructor is both a function and an object

```javascript
Function.prototype.isPrototypeOf(Point) // true
Object.prototype.isPrototypeOf(Point) // true
```
The **prototype** property

The building prototype exists only for constructors and defines properties for all the objects built by that constructor.

To define a specific building prototype you need to:

- define an object with desired properties playing the prototype role
- assign that object to the `prototype` property of the constructor

The **prototype** property can be dynamically changed but it affects only newly created objects.
Example (1/2)

Given the constructor

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

we want to associate a prototype to it so that `getX` and `getY` functions will be defined.

Note that the form `function Point() { ... }` does not make the `Point` identifier global, leading to problems if the prototype is added from an environment where `Point` is invisible.
Define the constructor for the object which will play the prototype role

```javascript
GetXY = function() {
    this.getX = function() { return this.x }
    this.getY = function() { return this.y }
}
```

Create it and assign it to the prototype property of the `Point` constructor

```javascript
myProto = new GetXY(); Point.prototype = myProto
```

You can invoke `getX` and `getY` on newly created `Point` objects only

```javascript
p4 = new Point(7, 8); alert(p4.getX())
```
Architecture

**BEFORE**

Constructor

- `__proto__`
- `prototype`
- `properties`

**AFTER**

Constructor

- `__proto__`
- `prototype`
- `properties`

prototype = building prototype

prototype

building prototype `myProto`

getX
getY
Searching properties

**Constructor**

- `__proto__`
- `prototype`
- `properties`

**Object**

- `__proto__`
- `specific properties for the object`

Searching order for properties using the `__proto__` property

- `prototype`
- `building prototype myProto
  - `getX`
  - `getY`
New experiments (1/2)
New experiments (1/2)

🔍 Searching for p4 identity

myProto.isPrototypeOf(p4) // true
GetXY.prototype.isPrototypeOf(p4) // true
Point.prototype.isPrototypeOf(p4) // true
Object.prototype.isPrototypeOf(p4) // true
Function.prototype.isPrototypeOf(p4) // false

🔍 Searching for myProto and GetXY identities

Point.prototype.isPrototypeOfOf(myProto) // true
Object.prototype.isPrototypeOfOf(myProto) // true
Function.prototype.isPrototypeOfOf(myProto) // false
Point.prototype.isPrototypeOfOf(GetXY) // false
Object.prototype.isPrototypeOfOf(GetXY) // true
Function.prototype.isPrototypeOfOf(GetXY) // true
Instead of associating a new prototype to an existing constructor, it is possible to add new properties to the existing constructor

```javascript
Point.prototype.getX = function() { ... }
Point.prototype.getY = function() { ... }
```

The two approaches are not equivalent

- A change in the existing prototype affects also existing objects
- A new prototype affects only objects newly created from then on
Example (1/2)

Given the constructor

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

we want to modify the existing prototype so that `getX` and `getY` functions will be included.

Note that those functions will work for existing objects and for objects created from then on.
Example (2/2)

- Create a first object
  
  ```javascript
  p1 = new Point(1, 2)
  ```

- The function `getX` is not supported
  
  ```javascript
  p1.getX // returns undefined
  ```

- Modify the existing prototype
  
  ```javascript
  Point.prototype.getX = function() { return this.x }
  Point.prototype.getY = function() { return this.y }
  ```

- Now `getX` works even on existing objects
  
  ```javascript
  p1.getX() // returns 1
  ```
Prototype-based inheritance

Chains of prototypes are the mechanism offered by JavaScript to support a sort of inheritance.

It is an inheritance between objects, not between classes as in object-oriented languages.

When a new object is created using `new`, the system links that object with the building prototype for the constructor used.

This is also true for constructors, which have `Function.prototype` as their prototype.
Expressing inheritance

To express the idea of a subclass Student inheriting from an existing class Person you need to:

1. explicitly link Student.prototype with a new Person object
2. explicitly change the constructor property of Student.prototype (which now would link the Person constructor) to make it reference the Student constructor
Example (1/2)

Base constructor

```javascript
Person = function(n, y) {
    this.name = n; this.year = y
    this.toString = function() {
        return this.name + ' was born in ' + this.year
    }
}
```

Derived constructor

```javascript
Student = function(n, y, m) {
    this.name = n; this.year = y; this.matr = m;
    this.toString = function() {
        return this.name + ' was born in ' + this.year + '
        and has matriculation ' + this.matr
    }
}
```
Example (2/2)

Setting the chain of prototypes

Student.prototype = new Person()
Student.prototype.constructor = Student

Test

function test() {
  var p = new Person("Andrew", 1965)
  var s = new Student("Luke", 1980, "001923")
  // displays: Andrew was born in 1965
  alert(p)
  // displays: Luke was born in 1980 and has
  matriculation 001923
  alert(s)
}
Inheritance: an alternative (1/2)

An alternative approach can be employed without touching prototypes: reusing by `call` the base constructor function, simulating other languages, e.g. the use of `super` in Java.

```javascript
Rectangle = function(a, b) {
    this.x = a; this.y = b
    this.getX = function() { return this.x }
    this.getY = function() { return this.y }
}
Square = function(a) {
    Rectangle.call(this, a, a)
}
```
Inheritance: “super” in constructors

Base constructor

Person = function(n, y) {
    this.name = n; this.year = y
    this.toString = function() {
        return this.name + ' was born in ' + this.year
    }
}

Derived constructor

Student = function(n, y, m) {
    Person.call(this, n, y); this.matr = m;
    this.toString = function() {
        return this.name + ' was born in ' + this.year + ' and has matriculation ' + this.matr
    }
}
Inheritance: “super” in methods

When prototypes are explicitly manipulated, the prototype property can be used to call methods defined in the base constructor.

```javascript
Student = function(n, y, m) {
  Person.call(this, n, y); this.matr = m
  this.toString = function() {
    return Student.prototype.toString.call(this) + ' and has matriculation ' + this.matr
  }
}
```

The `Student.prototype` is a `Person` object, so `call` calls the `toString` function of that object.
An alternative: "super" in methods

Avoiding the use of prototypes, it is necessary to explicitly exploit an object of the kind of the prototype to invoke the desired method.

```javascript
Student = function(n, y, m) {
  Person.call(this, n, y); this.matr = m
  this.toString = function() {
    return p.toString.call(this) + ' and has matriculation ' + this.matr
  }
}
```

The `p` object must be a `Person` object which must exist when the function is called, so that `call` calls the `toString` function of that object.
Using the `Student` and `Person` constructor setting explicitly the chain of prototypes, the following results are obtained with `p` a `Person` object and `s` a `Student` object:

- `p.isPrototypeOf(s)` // false
- `Person.isPrototypeOf(s)` // false
- `Object.isPrototypeOf(s)` // false
- `Object.prototype.isPrototypeOf(s)` // true
- `Person.isPrototypeOf(Student)` // false
- `Student.prototype.isPrototypeOf(Student)` // false
- `Student.prototype.isPrototypeOf(Student.prototype)` // false
- `Student.prototype.isPrototypeOf(s)` // true
Using the same environment as before, but without explicitly setting
the chain of prototypes, the following results are obtained:

```javascript
p.isPrototypeOf(s) // false
Person.isPrototypeOf(s) // false
Object.isPrototypeOf(s) // false
Object.prototype.isPrototypeOf(s) // true
Person.isPrototypeOf(Student) // false
(new Person()).isPrototypeOf(Student) // false
(new Person()).isPrototypeOf(Student.prototype) // false
(new Person()).isPrototypeOfOf(s) // false
```
Arrays (1/2)

- An array is built using the `Array` constructor, whose arguments are the initial content of the array
  
  ```javascript
  colors = new Array('red', 'green', 'blue')
  ```

- Elements are enumerated starting with 0 and can be accessed using square brackets, e.g. `colors[2]`

- The `length` attribute contains the dynamic length of the array

- Cells in an array are not constrained to contain elements of the same kind
Arrays (2/2)

It is also possible to define an empty array and add elements later using assignments

```javascript
colors = new Array(); colors[0] = 'red'
```

Starting with JavaScript 1.2, an array can be built listing the initial elements, separated by commas, between square brackets

```javascript
numbers = [1, 2, 'three']
```
Dynamic and fragmented arrays

- It is possible to dynamically add elements to arrays whenever it is necessary
  
  ```javascript
  letters = ['a', 'b', 'c']; letters[3] = 'd'
  ```

- Arrays can be fragmented: indexes have not to be in a set of adjacent numbers
  
  ```javascript
  letters[9] = 'j'
  ```

  ```javascript
  letters.length returns 10
  ```

  ```javascript
  letters.toString() returns a,b,c,d,,,,,,j
  ```
Objects as arrays (1/2)

- Every JavaScript object is defined by the set of its properties: this is why they are internally represented as arrays.

- This mapping between objects and arrays let object access be possible through an array-like notation using the property name as a selector.

- Let $p$ be an object, $s$ a string containing the name of the property $x$ of $p$; then the notation $p[s]$ gives access to the property named $x$ like the dot notation $p.x$ does.
Objects as arrays (2/2)

What is the advantage of the array notation over the dot notation?

Using the dot notation `p.x` implies that the name of the property is known when writing the program.

The array notation `p[s]` let the programmer access a property whose name can be known during execution and saved in the string variable `s` for future use.
Introspection

Since the set of an object’s properties can dynamically change, it may be necessary to discover which properties an object has at runtime.

A special construct is available to iterate on the visible properties of the object:

```javascript
for (variable in object) { … }
```

For example, to list the name of all properties:

```javascript
function showProperties(obj) {
    for (var p in obj) { document.write(p + ‘<br>’) }
}
```
From introspection to intercessation

Using the `for/in` construct it is possible to discover the visible properties of an object.

To access those properties you need to obtain a reference to them starting from a string containing the name of each property.

```javascript
function showProperties(obj) {
  for (var p in obj) {
    var property = obj[p]
    document.write('The property ' + p + ' has type ' + typeof(property) + '<br>
  }
}
```
The global object

JavaScript does not distinguish object methods from global functions: global functions are methods of a system-defined global object.

The global object features:

- as methods, functions not owned by specific objects and predefined functions
- as data, global variables
- as functions, predefined functions
Global predefined functions

eval – evaluate the JavaScript program passed as a string (reflection, intecession)

escape – convert a string in a portable format, substituting “illegal” characters with escaped sequences (e.g. `%20` for ` ' `)

unescape – convert a string from the portable format to the original format

isFinite, isNaN, parseFloat, parseInt, ...

...
(Constructors of) Predefined objects

Most common are Array, Boolean, Function, Number, Object, String

The Math object contains a mathematical library: constants (E, PI, LN10, LN2, LOG10E, LOG2E, SQRT1_2, SQRT2) and functions of all sorts

Don’t instantiate it: use it as a static component

The Date object contains features to represent date and time concepts and work with them

The RegExp object supports working with regular expressions
Date: construction (1/2)

Constructors

Date(), Date(milliseconds), ...

The Date() constructor creates an object representing current day and hour on the system in use.

In Date(milliseconds), milliseconds are calculated starting from 00:00:00 of January 1st, 1970, using the UTC standard day of 86.4M sec.
Constructors

Date(string), Date(year, month, day [, hh, mm, ss, ms])

UTC and GMT are supported

Days go from -100M to +100M around 1/1/1970

In Date(string), string must be in the format recognized by Date.parse

In Date(y, m, d), year, month and day must be provided; other parameters are optional; parameters not provided are set to 0
Date: methods

Methods

getDay returns the day of the week from 0 (Sunday) to 6 (Saturday)

getDate returns the day from 1 to 31

getMonth returns the month from 0 (January) to 11 (December)

getFullYear returns the year on four digits

gETCHours returns the hour from 0 to 23

getMinutes returns the minute from 0 to 59

getSeconds returns the seconds from 0 to 59

...
Date: example

Example

d = new Date(); millennium = new Date(3000, 00, 01)
s = new String((millennium - d) / 86400000)
days = s.substring(0, s.indexOf('.')) // integer part
alert(days + ' days to the year 3000')

Output (on March 5th, 2006)
362987 days to the year 3000
Who is the global object?

The global object is unique and it is always created by the interpreter before executing anything.

There is no global identifier: in every situation there is a given object used as global object.

- in a browser, that object is typically `window`.
- but on the server side, it would probably be another object to play the role of global object.

Could it be a problem not to know which object plays the role of global object?
The global object: warnings

- Function and variables not assigned to a specific object are assigned to the global object...
- ...but if they appear in a function's scope they are assigned as local to that scope.
- There are no problems, if global properties are used without making the global object emerge.
- There can be problems if `eval` or another reflexive function is used, since `eval("var f")` is different from `var f` because the first definition is not executed in the global environment.
Global object and functions as data (1/4)

JavaScript lets variables reference functions and functions be passed as arguments to other functions

```javascript
var square = function(z) { return z*z }

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

But the `f` variable

must reference a function object

cannot be a string containing the name of an already defined function

```javascript
exe("Math.sin", .8) // error
```
Global object and functions as data (2/4)

 Beside the approach based on the `Function` constructor, the global object can be exploited to obtain a reference to a function object corresponding to a given function name.

 Let `p` be a reference to an object, and `s` a string containing the name of the `x` property of `p`, then the array-like notation `p[s]` returns a reference to the property `x`.

 In this case, `p` is the global object, `s` a function name, `x` the function object corresponding to the name in `s`.
Global object and functions as data (3/4)

The following notation

```javascript
var name = Math[“sin”]
```

puts in the `name` variable a reference to the function object `Math.sin`

So, after defining the function

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

we can invoke

```javascript
exe(name, .8) // returns 0.7173560908995228
```

because the “sin” string has been translated into a reference to the `Math.sin` object, suitable for invocation
Global object and functions as data (4/4)

- Generalizing
  ```javascript
  var fun = prompt("Enter a function name")
  var f = Math[fun]
  ```

- Now the user can specify a function name and let it be searched and invoked by a reflexive mechanism.

- The result can be showed in another window
  ```javascript
  confirm("Result: " + exe(f, x))
  ```

- Note that in this example the Math object plays the role of the global object since functions are searched in it only
Forms and their management (1/3)

- JavaScript is often used in the context of HTML forms
- A form usually contains text fields and buttons

```html
<form name="aForm">
  <input type="text" name="textField" size="30" maxlength="30">
  <input type="button" name="button" value="Click here">
</form>
```

- When the button is pressed, it is possible to invoke a JavaScript function
Forms and their management (2/3)

When a button is pressed, the button pressed event can be intercepted by the `onclick` attribute

```html
<form name="aForm">
  <input type="button" name="button" value="Click here" onclick="alert('You clicked me!')">
</form>
```

Remember to alternate double and single quotes when writing JavaScript code in HTML attributes.
As an alternative example, when the button is pressed we can make the browser write the result of one of our functions

```html
<form name="aForm">
  <input type="button" name="button" value="Click here" onclick="document.write(square(6))">
</form>
```

Note that `square` must be already defined.
Forms: which events?

Events which can be intercepted on an element (managed on the correspondent tag)
- onclick, onmouseover, onmouseout, ...

Events which can be intercepted on a window (managed in the body tag)
- onload, onunload, onblur, ...

Example

```html
<body onload="alert('Loaded!')">
  <form name="aForm">
    <input type="button" name="button" value="Click here" onclick="alert(square(6))">
  </form>
</body>
```
To reuse the value returned by `confirm`, `prompt`, or other functions, a whole JavaScript program has to be inserted as the value of the `onclick` attribute (as a sequence or a function call).

**Examples**

```javascript
onclick="x = prompt('Name and surname');
document.write(x)"
```

```javascript
onclick="ok = confirm('Is this OK?'); if (!ok)
alert('Warning!')"
```
Forms and text fields

Text fields can be objects with a name within a form object with a name.

As such, they can be referenced using the dot notation, e.g. `document.aForm.aTextField`.

Text fields are characterized by the `value` property.

Example

```html
<form name="aForm">
  <input type="text" name="surname" size="20">
  <input type="button" name="button" value="Show" onclick="alert(document.aForm.surname.value)">
</form>
```
Functions as links

- A JavaScript function can be used as a valid link usable as the href attribute of the a element.
- The effect of a click on that link is the execution of the function and the display of the result in a new HTML page within the same window.
- Example
  
  `<a href="javascript:square(10)">This should be 100</a>`