CSI 3540

Structures, techniques et normes du Web

Le langage JavaScript (ECMAScript)

Objectif:

- Écrire des programmes JavaScript
- Introduction à l'environnement d'éxécution

Lectures:

- Web Technologies (2007) § 4

Plan

1. Motivation
2. Bref historique
3. Présentaton détaillée

1. Grammaire
2. **Objets**
3. Héritage par prototype
4. Fermetures

Mythe ou réalité

- **JavaScript** est

  1. un langage de script dont la syntaxe est dérivée de Java

  2. une version interprétée de Java (sans code intermédiaire, code-octet)

  3. un langage de programmation orienté prototype (sans classe)

Qu’en pensez-vous?

- Quelles sont vos connaissances de **JavaScript** ? Novice, intermédiaire, avancé ?
- Qu’en pensez-vous? C’est un excellent ou mauvais language? Ou peut-être, vous êtes sans opinion.

Réalité

- **JavaScript** est un langage de programmation avec d'**excellentes assises** et quelques mauvais concepts
- Victime de son adoption rapide
- Avec un peu de **discipline**, on se limite aux excellents concepts offerts
- Trop souvent, le programmeur ne se donne pas la peine d'apprendre le langage

Réalité

- **Paradigmes JavaScript importants** pour la programmation d'applications Web
  - Objet, objet **littéral**
  - **Fonctions**, programmation fonctionnelle
  - **Fermeture**

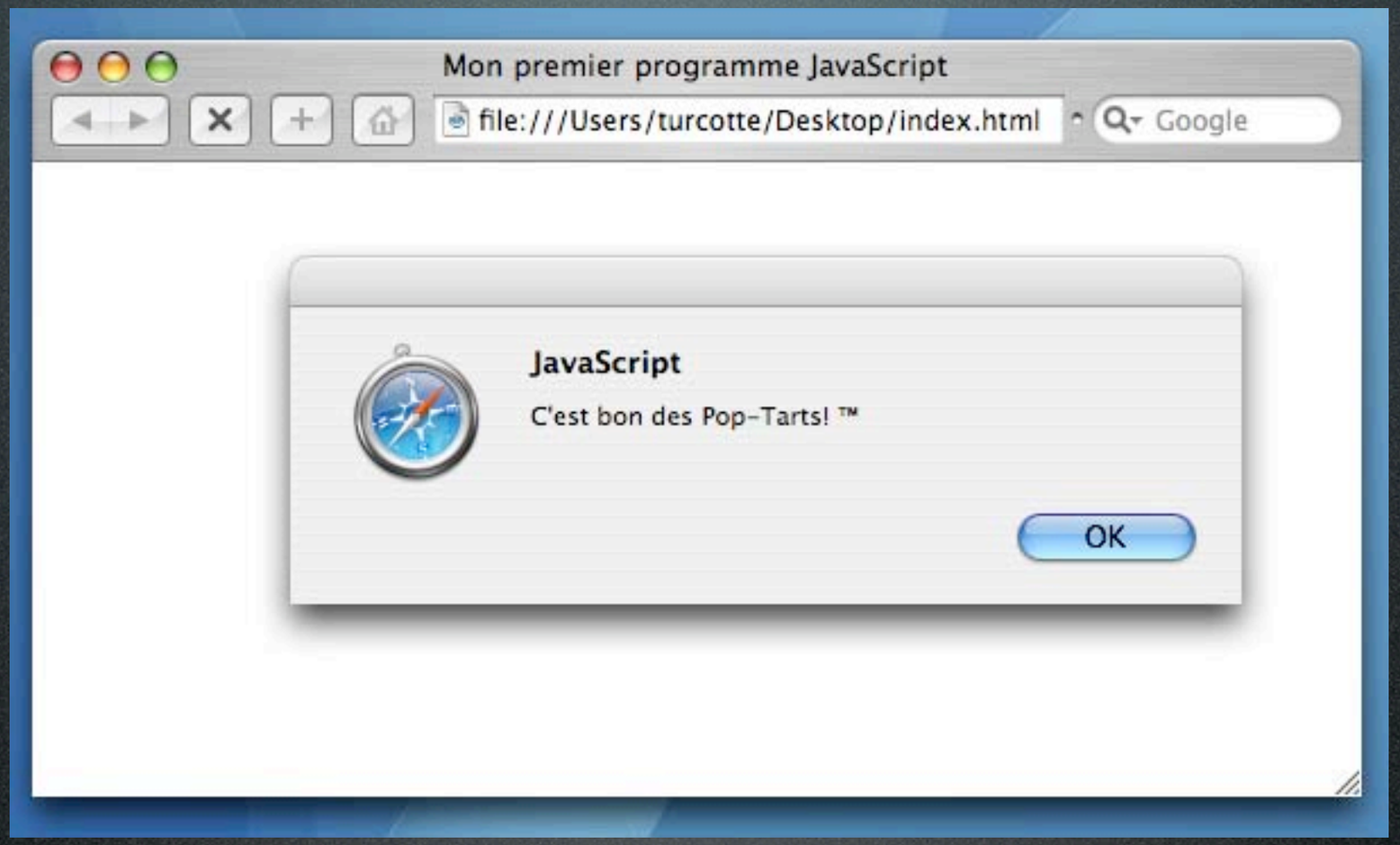
Motivation

- **Calcul côté client**
  - Par exemple, valider le format d'un numéro de téléphone, de carte de crédit, etc.
  - Réduire la charge du réseau et/ou du serveur
- **Construire les pages côté client**

Motivation

- Le modèle objet de document (**DOM** HTML) fournit un accès programmatique à la page courante de l'agent utilisateur
- **JavaScript** est un langage de programmation fréquemment utilisé pour manipuler le DOM
- **Concept important : paradigme pour les objets !**

```
<!DOCTYPE html
      PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-CA">
  <head>
    <title>Mon premier programme JavaScript</title>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
    <link rel="stylesheet" type="text/css" href="default.css" media="all" />
    <script type="text/javascript" src="HelloWorld.js">
    </script>
  </head>
  <body>
  </body>
</html>
```




```
alert( "C'est bon des Pop-Tarts! \u2122" );
```

```
<!DOCTYPE html
      PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-CA">
  <head>
    <title>Mon premier programme JavaScript</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" href="default.css" media="all" />
    <script type="text/javascript">
      alert( "C'est bon des Pop-Tarts! ™" );
    </script>
  </head>
  <body>
  </body>
</html>
```

Environnement de développement

- Supporté par virtuellement tous les fureteurs, c'est donc l'un des langages de programmation les plus utilisés
- Rhino – JavaScript for Java [http://www.mozilla.org/rhino] 2007

```
$ java -jar js.jar
Rhino 1.7 release 2 2009 03 22
js>
```

La p'tite histoire de JavaScript

- Né **LiveScript** (nom de code Mocha) en 1995. Créé par Brendan Eich pour Netscape 2.0
- Microsoft crée **JScript** en juillet 1996
- 1997, un standard est proposé par European Computer Manufacturers Association (ECMA). **ECMAScript** ECMA-262
- 1999, ECMA-262 3ième édition
- 5ième édition adoptée en décembre 2009

Hum?

- Contrairement à ce que le nom suggère, JavaScript n'est pas un descandant de Java!
- En fait, on devrait plutôt parler d'ECMAScript 8-(

Langages de scripts

- Langages de programmation **généralement interprétés**, à **typage dynamique**, et qui servent à l'**extension de systèmes** (accès programmatique aux structures internes d'un système)
- Les **langages de commandes**, tels que Bash et C-Shell, sont une extension du système d'exploitation
- Le langage **TCL** peut être intégré à tout logiciel afin de le rendre extensible

JavaScript

- **Rends les clients et serveurs extensibles** en donnant un accès programmatique aux structures internes
- Interprété
- Typage dynamique
- Syntaxe semblable à celle de Java et C++

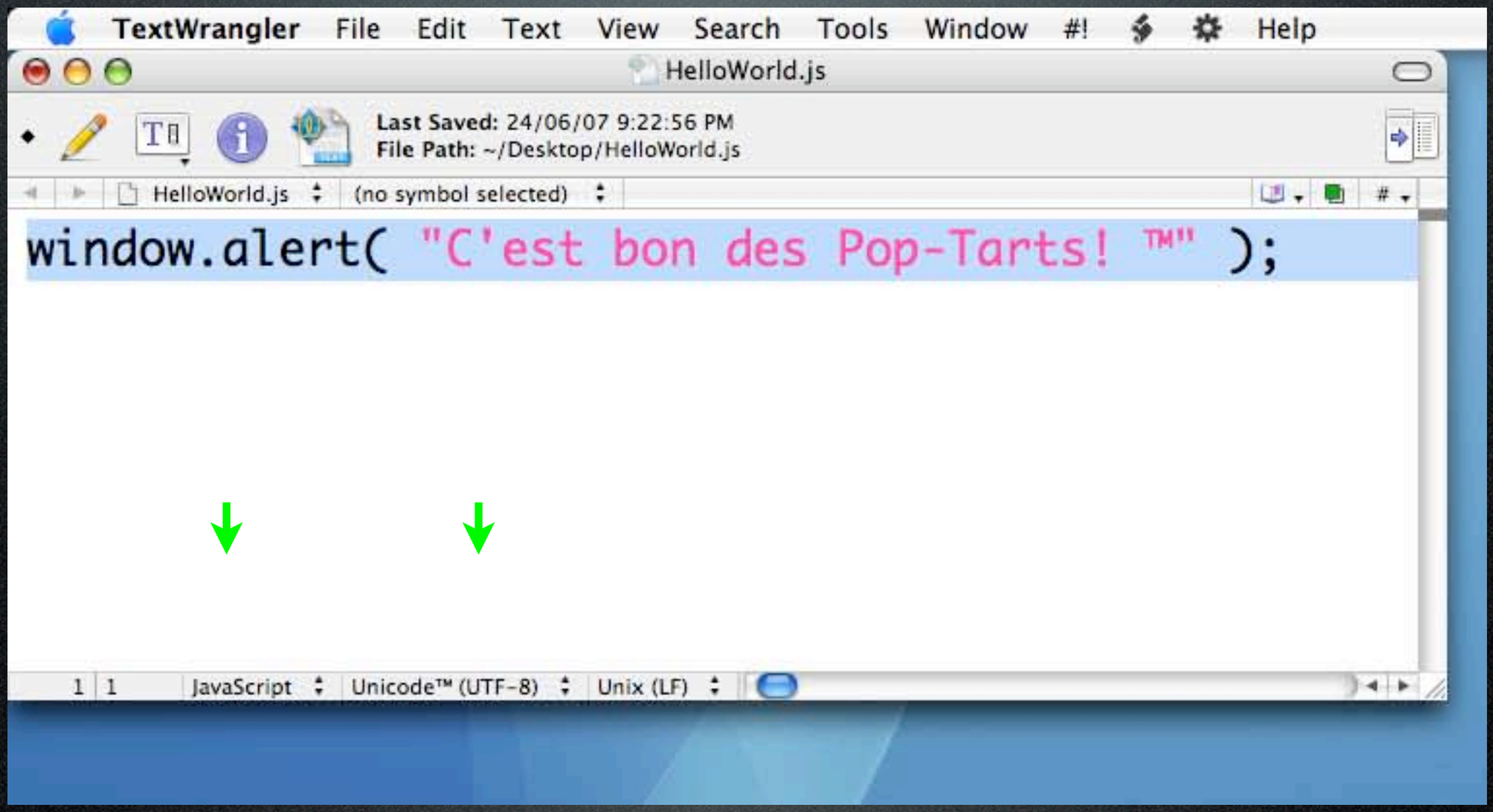
Machine virtuelle
+ runtime
Cette présentation
ne couvre que
'Scripting engine'

JavaScript

Comprend 2 parties :

- **Machine vituelle** (scripting engine) : interprète et fonctions essentielles d'ECMAScript (présentée aujourd'hui)
- **Environnement hôte** (hosting environment) : spécifique au client (Mozilla, IE, etc.), au server (IIs, etc.)...

Édition

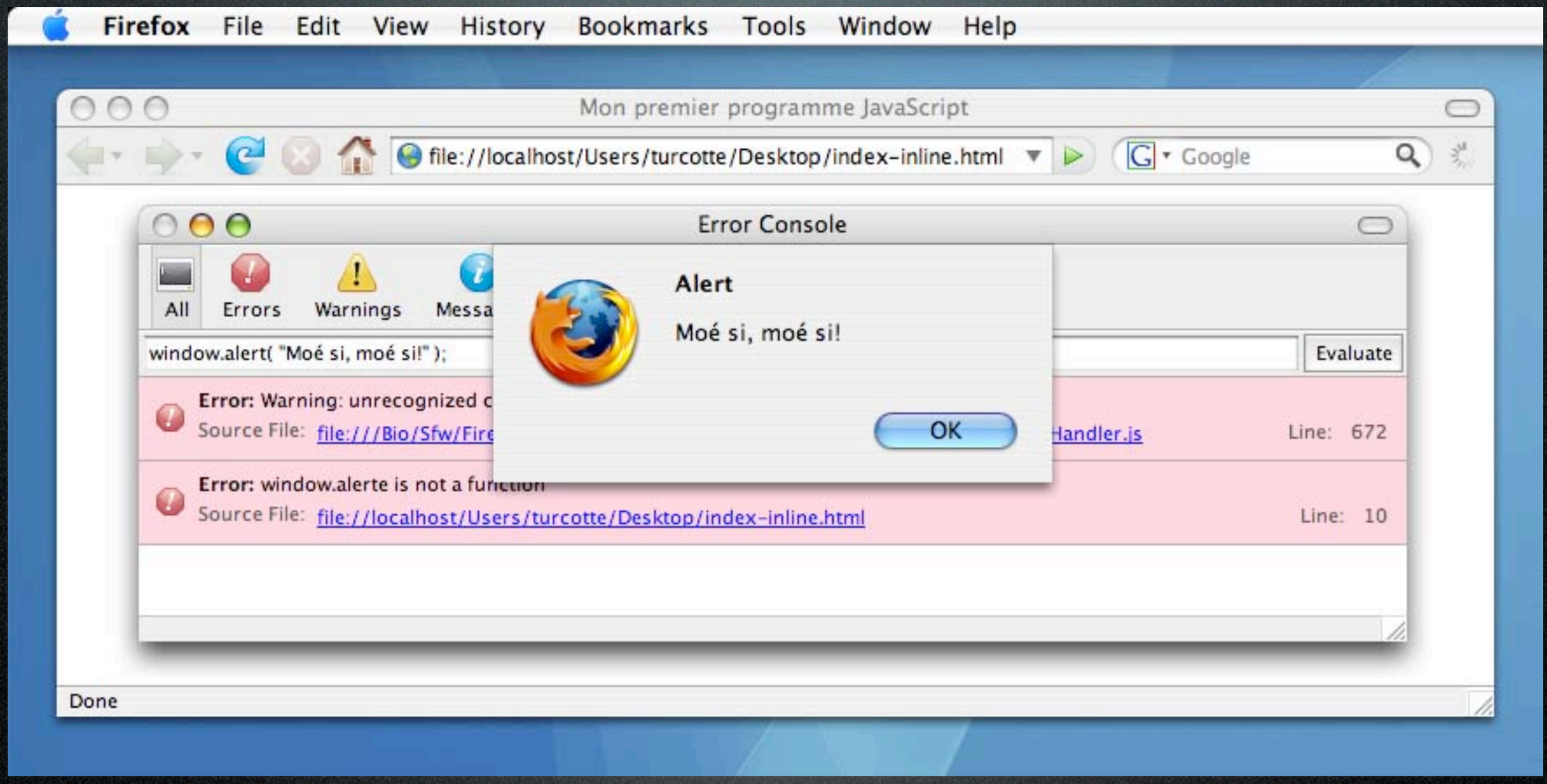

TextWrangler File Edit Text View Search Tools Window #! Help
HelloWorld.js
Last Saved: 24/06/07 9:22:56 PM
File Path: ~/Desktop/HelloWorld.js
HelloWorld.js
(no symbol selected)
window.alert("C'est bon des Pop-Tarts! ™");
1 1 JavaScript Unicode™ (UTF-8) Unix (LF)

«Rhino: JavaScript for Java»

- Implémentation code source libre en Java
- www.mozilla.org/rhino/
- Comprends l'API, un shell, un débogueur, un compilateur, une documentation, etc.

```
$ java -jar js.jar
Rhino 1.7 release 2 2009 03 22
js> 1 + 1
2
```

Mise au point



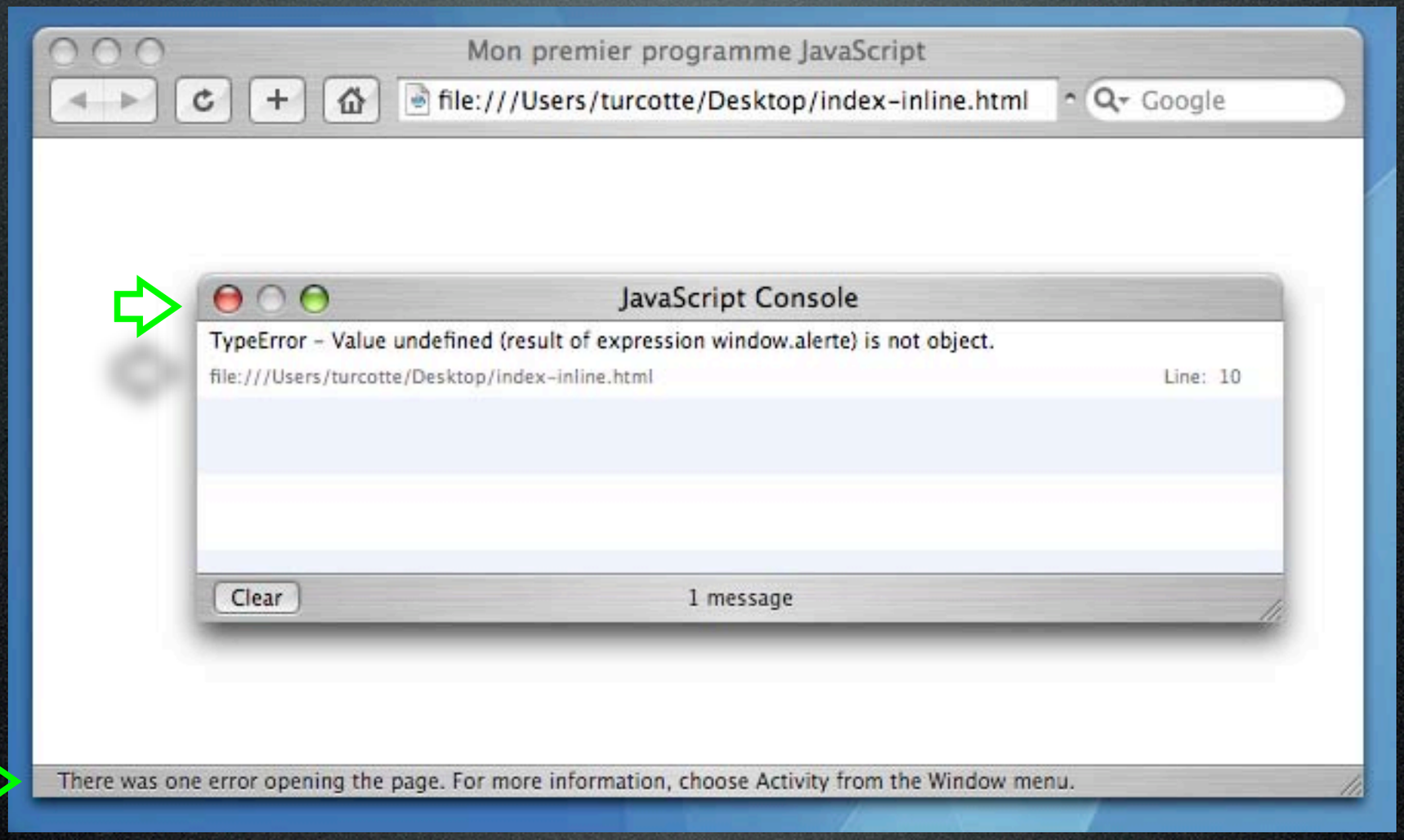
IE6

- ‘(…) I suggest selecting **Tools|Internet Options…** from the IE6 menu and click on the Advanced tab in the pop-up window that appears. Then check the “Display a notification about every script error” checkbox.’ [Jackson (2007) page 197]

Mise au point

- **Venkman** est le débogueur de Mozilla, qui tourne entre autres sous Firefox

Exécution



> defaults write com.apple.Safari IncludeDebugMenu 1

Le monde d'ECMAScript

- **SpiderMonkey** est l'implémentation code source libre écrite en C de Mozilla (embarquée dans Firefox)
- **Tamarin** est un engin (une machine virtuelle) JavaScript, haute performance, code source libre, écrit en C++ (Projet Mozilla, embarqué dans Adobe Flash Player)

Language
Engine
Applications
Hook/Convertor
Company
Implementation Language
Microsoft
Silverlight
JScript.NET
Internet Explorer
JScript
.NET Framework
JScript
DMDScript
QtScript
DMD
Qt Toolkit
iCab
InScript
iCab
Konqueror
KJS
Opera
futhark
linear_b
Opera
PDF
Firefox
Camino
ActionMonkey
Spidermonkey
C/C++
Flash on C++
Adobe
Flex
Flash
ActionScript
Photoshop
ExtendScript
AIR
Tamarin
JSCore
Safari
Apple
Android
python-spidermonkey
Python
IronPython
IronRuby
RubyJS
Mozilla
Narcissus
JavaScript
ruby-spidermonkey
Ruby
Rhino
GWT
Java
Google
Phobos
Heima
Tomcat
CouchDB
HD DVD
mod_js
mod_gcj
mod_jk
Apache
ParenScript (Lisp)
YHC/JavaScript
Scheme2JS
Haskell
Haxe (OCaml)
Scheme
JavaScript:: Spidermonkey
mod_perl
Perl

The World of ECMAScript

By John Resig (ejohn.org)
Updated November 15, 2007
Released under the GPL v2

JavaScript

Le langage de programmation

Similitudes avec Java

Grammaire

- La syntaxe du language (affectations, opérateurs, blocs et structures de contrôles) ressemble à celle de Java
- Les tableaux sont des objets
- Sensible à la casse (case sensitive)
- Commentaires : // et /* */

Dissimilitudes par rapport à Java

Types de données

- Les **types sont associés aux valeurs** et non pas aux variables! Ces types sont :
  - **Number**
  - **String**
  - **Boolean**
  - **null**
  - **undefined**
  - **Object**

Types de données

```
<script type="text/javascript">
  var v, m1, m2, m3, m4, m5;
  m1 = typeof v;
  m2 = typeof w;
  v = null;
  m3 = typeof v;
  v = 99;
  m4 = typeof v;
  v = m2;
  m5 = typeof v;
  v = true;
  m6 = typeof v;
  alert( m1 + "," + m2 + "," + m3 + "," + m4 + "," + m5 + "," + m6 );
</script>
```

La variable v reçoit des valeurs de divers types.
L'opérateur typeof
Notez la différence entre undefined et null

Types de données

- **Number**, **String** et **Boolean** sont **immuables** et **possède des méthodes**
  - p.e. `"informatique".toUpperCase()`
- Les **tableaux** et les **fonctions** sont des objets

Types de données: Number

- **Aucune distinction entre les nombres réels et les nombres entiers**
- Représention 64 bits point flottant (IEEE 754)
- **NaN** et **isNaN(v)**, **+Infinity**, **-Infinity**

Conversions

- Plusieurs **conversions de type** automatiques

Conversion : Boolean

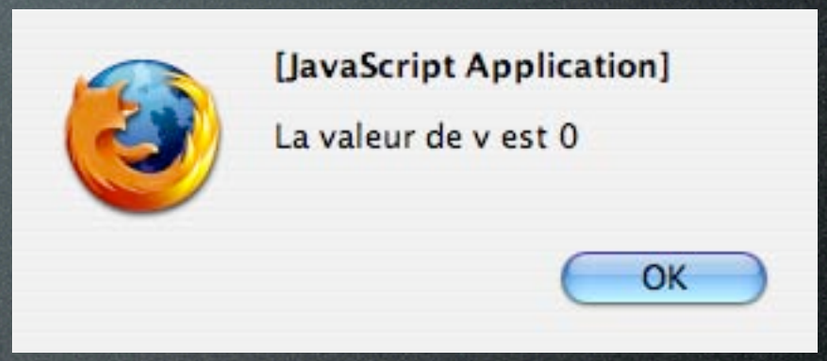
Originale	Booléenne
undefined	false
null	false
0	false
NaN (‘Not a Number’)	false
Infinity, -Infinity	false
“”	false
Toute autre valeur	true

Originale	String
undefined	undefined
null	null
true, false	true, false
NaN	NaN
Infinity, -Infinity	Infinity, -Infinity
Petits nombres	Notation décimale
Grands nombres	Notation scientifique
Objet	Appel à **toString**()

Conversion : Number

Originale	**Number**
undefined	NaN
null, false, “”	0
true	1
Chaîne représentant un nombre	Valeur du nombre
Autres chaînes	NaN
Objet	Appel à **valueOf()**

Conversion automatique des types de valeurs




```
var v = 10;
while ( v ) {
  v = v - 1;
}
alert( "La valeur de v est " + v );
```

L'exemple démontre aussi l'initialisation d'une variable dans le contexte de sa déclaration

Syntaxe

- **Variables** : précédées par le mot clé **var** et n'ont aucune déclaration de type **var id**;
- Les identificateurs débutent par une lettre ou un soulignement
- Le reste est constitué de lettres, chiffres et soulignements
- Ne sont pas des mots réservés

Syntaxe

- Une variable sera **automatiquement créée** lors d'une **affectation** si nécessaire

```
$ js
Rhino 1.6 release 7 2007 08 19
js> var x = 1;
js> y = 2;
2
js> x + y
3
js> x + z
js: "<stdin>", line 5: uncaught JavaScript runtime exception:
ReferenceError: "z" is not defined.
```

Portée

- **Environnement global** (variables globales)
- **Variables locales** sont déclarées à l'aide de **var**, mais un bloc d'énoncés ne crée pas un nouvel environnement lexical

Syntaxe

- L'énoncé **throw** : throw("Sans objet ;-)")
- Le bloc **catch** :

```
try {
        ...

} catch ( e ) {
   window.alert( "J'attrappe tout : " + e );
}
```

Syntaxe

- throw({ name : "FormatException",
  message : "2EC3 is not a valid number" })

Autres

- Pas de **main(String[] args)**;

Fonctions

Première partie

Fonctions

- Mot clé function introduit la déclaration d'une fonction
- Les paramètres formels n'ont **pas de déclaration de type**
- Ces fonctions **n'appartiennent pas à un objet** ou une classe
- Appels de fonction **par valeur** ; comme les appels de méthodes de Java

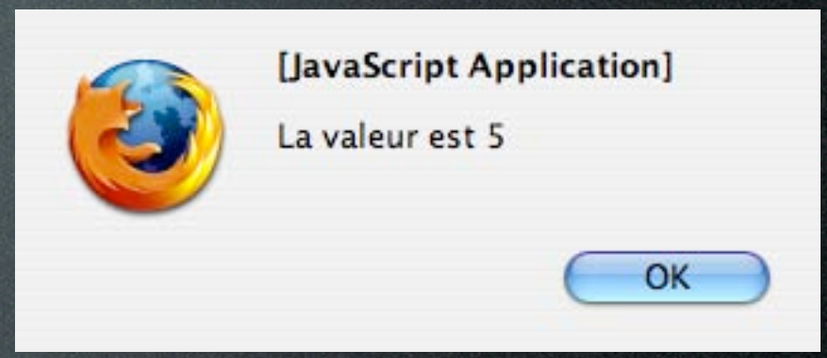
```
function maximum( a, b ) {
   if ( a > b ) {
      return a;
   } else {
      return b;
   }
}

function minimum( a, b ) {
   if ( a < b ) {
      return a;
   } else {
      return b;
   }
}

var x = 5, y = 10, z = maximum( x, y );
```

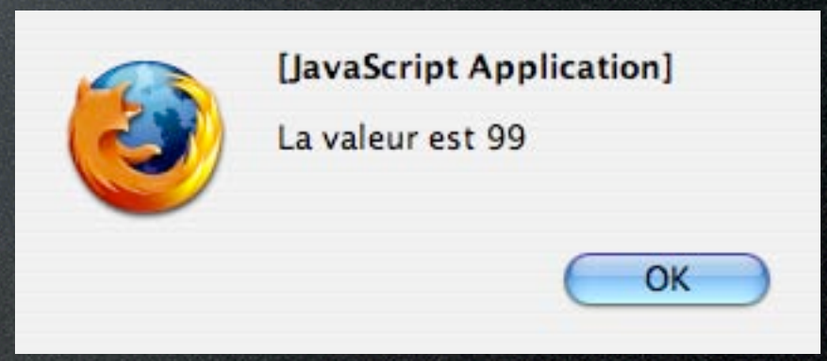
Fonctions

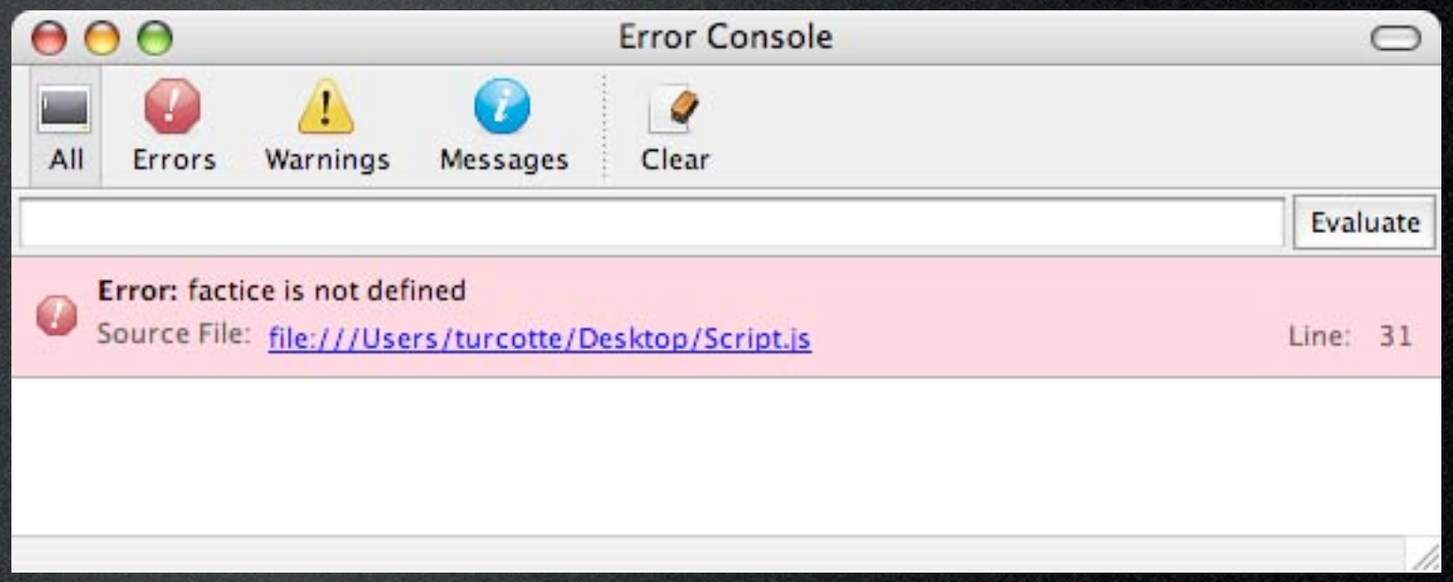
- Attention, si le dernier énoncé exécuté n'est pas un énoncé de retour (return) alors la valeur de retour est **undefined**

Valeur de type fonction

```
function affiche( f, a, b ) {
   alert( "La valeur est " + f( a, b ) );
}

affiche( minimum, x, y );

fun = function factice( a, b ) {
   return 99;
};

affiche( fun, 5, 10 );

affiche( factice, 5, 10 );
```

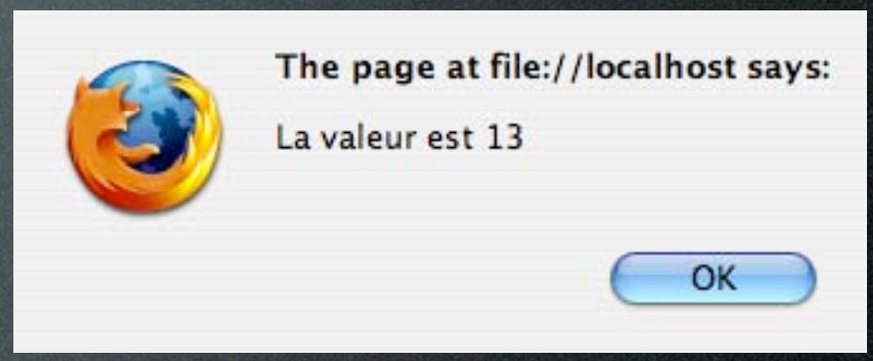






Fonctions anonymes

```
function affiche( f, a, b ) {
    alert( "La valeur est " + f( a, b ) );
}

affiche( function( a, b ) { return a + b }, 6, 7 );
```



Fonctions prédéfinies

- eval
- isFinite, isNaN, parseInt, parseFloat
- encodeURI, decodeURI, encodeURIComponent, et decodeURIComponent

Très permissif

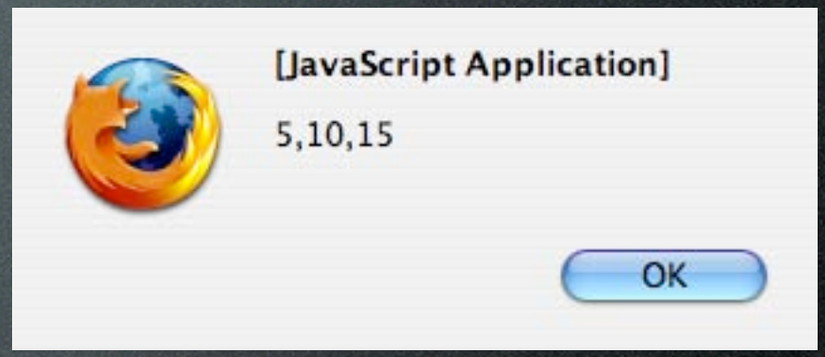
- Conversions de type automatiques
- Plusieurs «;» optionnels
- Paramètres surnuméraires ignorés lors des appels de fonction
- Affectation de la valeur undefined aux paramètres manquants lors des appels de fonction
- L'ordre des déclarations de fonctions n'a pas d'importance : l'exécution se fait en 2 passes

arguments

```
var somme = function() {
  var i, somme = 0;
  for ( i=0; i<arguments.length; i += 1 ) {
    somme += arguments[ i ];
  }
  return somme;
};

somme( 1, 2, 3 );
```

Variables locales, globales et appels par valeur

```
var a = 5, b = 10, c = 15;

function test( d ) {
  var a;

  a = 96;
  b = 97;
  c = 98;
  d = 99;

  alert( a + "," + b + "," + c + "," + d );
}

alert( a + "," + b + "," + c );

test( c );

alert( a + "," + b + "," + c );
```



Objets sans classe(s)

Programmation orientée prototype

Définitions

- L'**objet** en JavaScript est constitué d'un ensemble de **propriétés**
- Une **propriété** est composée d'un **nom unique** et d'une **valeur**
- Aucune restriction quant aux noms des propriétés, la chaîne vide est un nom de propriété admissible

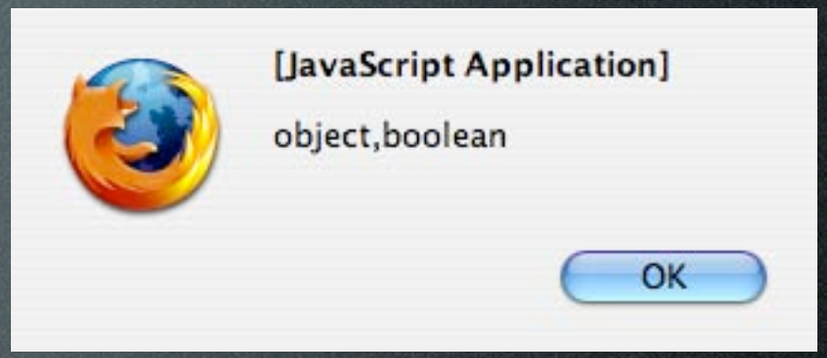
Remarques

- Tout comme les variables, **les propriétés** n'ont pas de type, elles **ont des valeurs**, et ces dernières ont un type
- Il s'agit des 6 types vus précédemment : **Number**, **String**, **Boolean**, **null**, **Object** et **undefined**

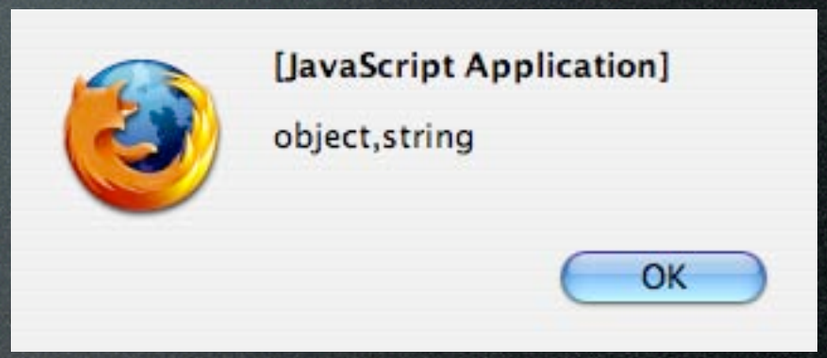
Remarques (suite)

- Les propriétés sont **ajoutées** ou même **retranchées** lors de l'**exécution** des programmes
- Il n'y a pas de déclarations statiques des objets, les objets sont créés **dynamiquement**, et les propriétés sont ajoutées dynamiquement
- Pas de classes

```
var o;
o = new Object();
o.prop = true;
window.alert( (typeof o) + "," + (typeof o.prop) );
```

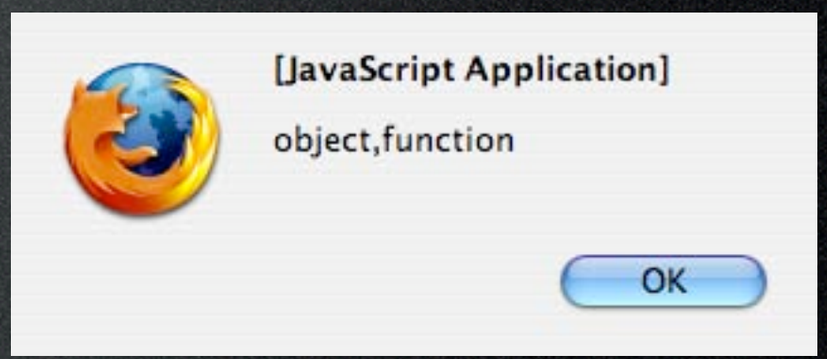



```
o.prop = "Ça pas de classe c't'affaire là";
window.alert( (typeof o) + "," + (typeof o.prop) );
```

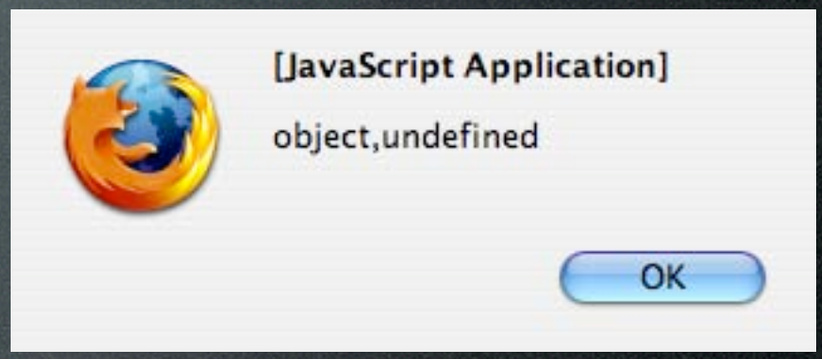



```
o.prop = 1;
window.alert( (typeof o) + "," + (typeof o.prop) );
```

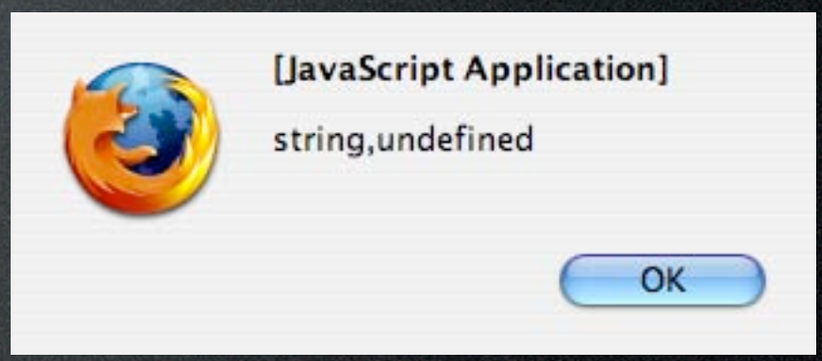

```
o.prop = function( x ) { return x + 1 };
window.alert( (typeof o) + "," + (typeof o.prop) );
```



```
delete o.prop;
window.alert( (typeof o) + "," + (typeof o.prop) );
```




```
o = "Alpha";
window.alert( (typeof o) + "," + (typeof o.prop) );
```



C'est un seul et même exemple

Sous Rhino

```
js> var o;
js> o = new Object();
[object Object]
js> o.prop = "Ça pas de classe c't'affaire là"
Ça pas de classe c't'affaire là
js> typeof o
object
js> typeof o.prop
string
js> o.prop = 1;
1
js> typeof o.prop
number
js> o = "alpha"
alpha
js> typeof o
string
js> typeof o.prop
undefined
```

Raccourci : « object initializers »

```
var personne = { prenom:"Gédéon", nas:1 2345, association:null };

var personne = { prenom:window.prompt( "Entrez le prénom" ),
                 nas:window.prompt( "Entrez le NAS" ),
                 association:null };

var obj = { };
```

Propriétés de l'objet

- Utilise des "" si le nom n'a pas le format d'un identifiant ou encore est un mot réservé

```
var stat = { "var": 2, "écart type": 1.41 , "valeur-associée": null };
```

```
var vol = { compagnie: "Air Canada",
            numéro: 342 ,
            depart: {
               ville: "YOW",
               heure: "16:12"
            }
          };
```

Réflection

- Mot clé in :

```
for ( var propr in personne ) {
    window.alert( propr );
}
```

Cette boucle affichera les (noms des) propriétés de l'objet désigné par la variable personne, soit prénom, nas et association (l'ordre n'est cependant pas défini)

Sous Rhino

```
js> var personne = { prenom: "Gédéon", nas:1 2345, association:null };
js> personne
[object Object]
js> for ( var propr in personne ) { print( propr ) };
association
nas
prenom
```

Propriétés d'un objet

- La variable propr contient le nom d'une des propriétés de l'objet personne,
- par exemple la valeur de la variable propr est "nom",
- **comment accèder à la valeur d'une propriété dont le nom est la valeur associée à une variable** ?

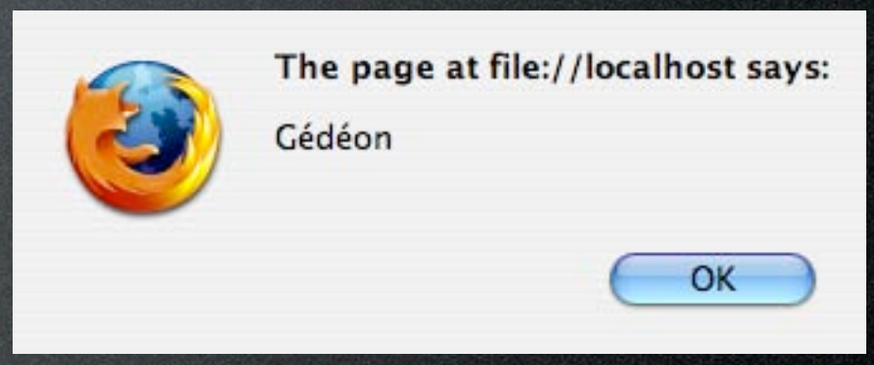
Propriétés d'un objet

- La notation personne.propr fait référence à une propriété dont le nom est propr
- Nous voulons accéder la propriété dont le nom est sauvegardé dans la variable propr
- Il faut utiliser la notation suivante : personne[propr]

Propriétés d'un objet

- Les deux appels à la méthode alert produiront le même résultat :

```
var propr = "prénom";
var personne = { prénom:"Gédéon", nas:1 2345, association:null };
window.alert( personne.prénom );
window.alert( personne[ propr ] );
```



- La notation personne[propr] est semblable aux tableaux associatifs et tables de hashage

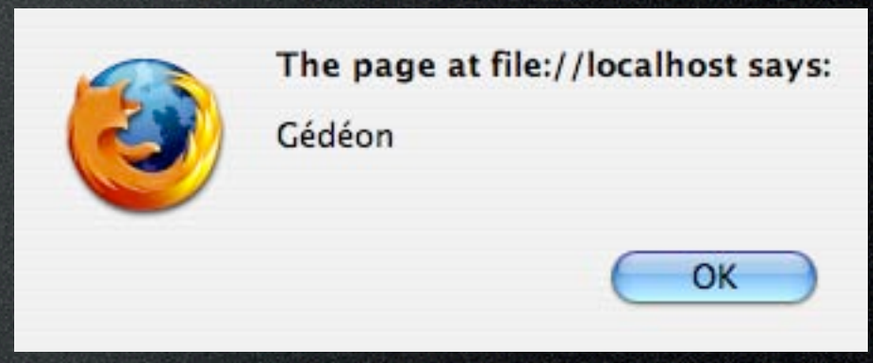
Références

```
js> var o1 = new Object();
js> o1.data = "Programmation";
Programmation
js> var o2 = o1;
js> o2.data += " Web";
Programmation Web
js> o1.data;
Programmation Web
```

Méthodes

- Les méthodes sont tout simplement des fonctions assignées aux propriétés
- Les deux appels à la méthode alert produiront le même résultat :

```
var personne = { prénom:"Gédéon",
                 getPrénom:function() { return this.prénom } };

window.alert( personne.prénom );
window.alert( personne.getPrénom() );
```



Conversion de types (suite)

```
js> o = { v : 10, valueOf : function() { return this.v } }
[object Object]

js> o.valueOf()
10

js> o + 1
11
```

Méthodes (suite)

- De même, les deux déclarations de méthodes suivantes sont (±) équivalentes :

```
var personne = { prénom:"Gédéon",
                 getPrénom:function() { return this.prénom } };
```

```
var f = function() { return this.prénom }
var personne = { prénom:"Gédéon", getPrénom:f };
```

- La déclaration du bas crée aussi une variable globale (qu'on ne peut utiliser directement, d'ailleurs)

Constructeur

```
function Tuple( first, second ) {
   this.first = first;
   this.second = second;

   this.getFirst = function() {
     return this.first;
   };
   this.getSecond = function() {
     return this.second;
   };
}

var p1 = new Tuple( "tree", "green" );
var p2 = new Tuple( "sky", "blue" );

window.alert( p1.getFirst() );
window.alert( p2.getSecond() );
```

- Sans new Object()
- Sans valeur de retour
- **Toute fonction est aussi un constructeur!**
- js> Tuple instanceof Object
  true
- new Tuple(...);
- p1 instanceof Tuple
- Object est un constructeur prédéfini

```
function Tuple( first, second ) {

   this.first = first;
   this.second = second;

   this.getFirst = function() {
          return this.first;
   };

   this.getSecond = function() {
          return this.second;
   };

   this.isOrdered = function() {
          return this.getFirst() <  this.getSecond();
   };

   this.swap = function() {
     var tmp = this.first;
     this.first = this.second;
     this.second = tmp;
   };
 }
```

```
var p1 = new Tuple( "tree", "green" );

p1.isOrdered();
// retourne false

p1.swap();
p1.isOrdered();
// retourne true
```

- En JavaScript, l'utilisation de this est toujours nécessaire, sinon le nom fait référence à une variable ou une fonction

- Notez la déclaration d'une variable locale dans la méthode swap

```
function makeTuple( first, second ) {

    var t = new Object();
    t.first = first;
    t.second = second;

    t.getFirst = function() { return this.first; };
    t.getSecond = function() { return this.second; };

    return t;
}
```

```
var t1 = makeTuple( "rouge", 1 );
print( t1.getFirst() );
```

```
function Tuple( first, second ) {
    this.first = first;
    this.second = second;

    this.getFirst = function() { return this.first; };
    this.getSecond = function() { return this.second; };
}

var t2 = new Tuple( "bleu", 2 );
print( t2.getFirst() );
```

__defineGetter__

```
function Time( h, m ) {

    this.totalMinutes = 60 * h + m;

    this.before = function( other ) {
        return this.totalMinutes < other.totalMinutes;
    }

    this.__defineGetter__( "hours", function() { return toInteger( this.totalMinutes / 60 );  } );
    this.__defineGetter__( "minutes", function() { return this.totalMinutes % 60;  } );
}

var t1 = new Time( 14, 30 );
var t2 = new Time( 15, 50 );

print( t1.before( t2 ) );

print( t1.hours );
print( t2.hours );

print( t1.minutes );
print( t2.minutes );
```

Similairement __defineSetter__ existe

Propriétés de l'objet

- toString(), valueOf()

Héritage

À l’aide de prototypes

Langages orienté protype

- Borning, A. H. The Programming Language Aspects of **ThingLab**, A Constraint-Oriented Simulation Laboratory. In ACM Transactions on Programming Languages and Systems, 3, 4 (**1981**) 353-387.

- Mais aussi **Self**, **Lisaac**, **Lua**, **Io**, **ABCL**...

Prototype

- Un **objet** sert de protype pour la création d'autres objets
- Un prototype est un **objet**
- Un objet est le **clone** d'un autre objet (prototype)

```
function Point() {
    this.x = 0;
    this.y = 0;
    this.getX = function() { return this.x };
    this.getY = function() { return this.y };
};

Object.create = function( obj ) {
    var Constructor = function() {};
    Constructor.prototype = obj;
    return new Constructor();
};

var p1  = new Point();
var p2 = Object.create( p1 );

p1.x = 3;
p2.x = 15;

print( p1.getX() );
print( p2.getX() );
```

- Affiche 3 et 15

Héritage (de prototype)

```
function Shape( x, y ) {
    this.x = x;
    this.y = y;

    this.getX = function() { return this.x };
    this.getY = function() { return this.y };
}

function Circle( x, y, r ) {

    this.x = x;
    this.y = y;
    this.r = r;

    this.getR = function() { return this.r };
}

Circle.prototype = new Shape();
```

- En Java, on aurait écrit «**class Circle extends Shape**»
- **Héritage simple ou multiple?**
- **Héritage simple**, un seul prototype associé

```
function ColoredCircle( x, y, r, c ) {

    this.x = x;
    this.y = y;
    this.radius = r;
    this.color = c;

    this.getColor = function() { return this.color };
}

ColoredCircle.prototype = new Circle();

var cl  = new ColoredCircle( 50, 100, 15, "bleu" );

js> for ( var p in cl  ) { print( cl[ p ] ); }
y
getColor
color
x
radius
getRadius
getX
getY
```

```
function EmptyStackException() {
    this.message = "Stack is empty";
}
EmptyStackException.prototype = new Error( "IllegalState" );

function Stack() {
    // ...
    this.pop= function() {
        if ( this.isEmpty() ) {
            throw new EmptyStackException();
        }
        // ....
    }
}
```

```
try {
    s.pop();
} catch ( e ) {
    if ( e instanceof EmptyStackException ) {
     print( "exception caught: " + e );
    }
}
```

Héritage (de prototype)

- L'héritage est donc dynamique
- Un objet peut changer de parent dynamiquement...
- Comment déterminer si la propriété appartient à l'objet ou est héritée?

```
for (propr in obj) {
    if ( obj.hasOwnPerperty( propr ) ) {
        ...
    }
}
```

Tableaux

Tableaux

```
var tbl1 = new Array();

var tbl2 = new Array( 99, "Okay", true );

tbl2[ 0 ];
// retourne la valeur 99

tbl2.length;
// retourne 3

tbl2.couleur = "Bleu";
tbl2.couleur;
// retourne "Bleu"

tbl2.length;
// retourne 3!

var tbl3 = [ 99, "Okay", true ];
```

```
var mat = [ [ 1, 0 ], [ 0, 1 ] ];
mat[ 0 ][ 0 ];
// retourne 1

mat[ 0 ][ 1 ];
// retourne 0

tbl2[ 3 ] = -1.0;
tbl2.length;
// retourne 4

tbl2.length = 2;

var tbl4 = new Array( 10 );
tbl4.length;
// retourne 10 !!!!

typeof tbl4[ 0 ];
// retourne undefined !!!!
```

Tableaux

```
var tbl = new Array( 5 );

tbl[ 0 ] = new Tuple( "Shabana", 1 );
tbl[ 1 ] = new Tuple( "Ashour", 2 );
tbl[ 2 ] = new Tuple( "Palmer", 3 );
tbl[ 3 ] = new Tuple( "Gaultier", 4 );
tbl[ 4 ] = new Tuple( "Lincou", 5 );
```

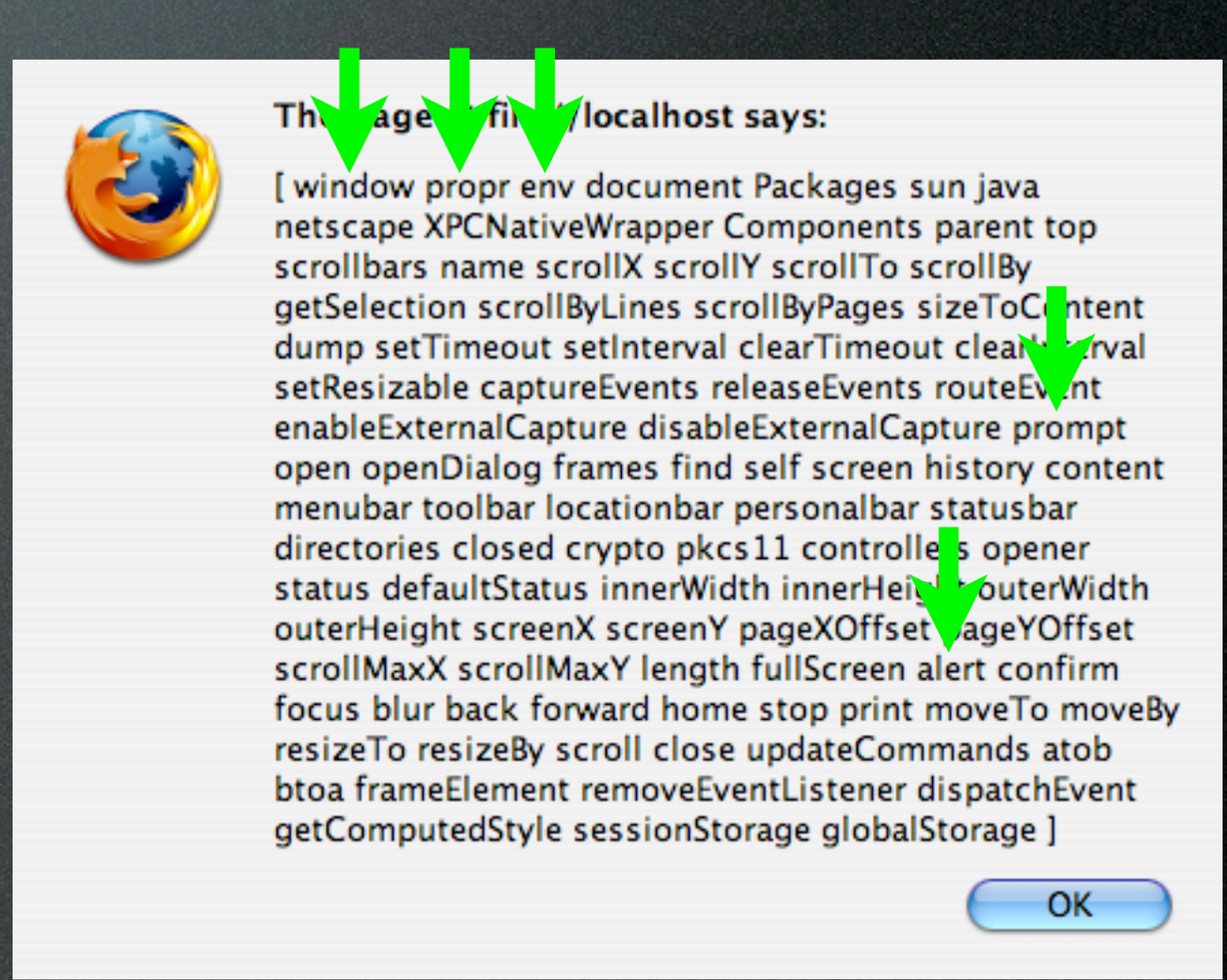
- toString(), splice(...), push(x), pop(), shift()

```
tbl.sort( function( a, b ) {
   if ( a.getFirst() <  b.getFirst() )
      return -1;
   else if( a.getFirst() == b.getFirst() )
      return 0;
   else
      return 1;
} );
```

Objets prédéfinis

Objet global (window)

```
var env = "[";

for ( var propr in window ) {
    env = env + " " + propr;
}

env = env + " ]";

window.alert( env );
```



Quelques autres objets prédéfinis

- L'environnement global est un objet
- Les variables globales ne sont que des propriétés de l'objet global
- L'environnement contient des objets prédéfinis
  - **Number**, **String** et **Boolean**
  - **Date**, **Math** et **RegExp**

Quelques autres objets prédéfinis

- Dans un navigateur, l'objet global s'appelle **window**
- Il fournit un accès programmatique aux scripts
- Entre autres, la propriété **window.document** représente le document HTML correspondant à la page courante
- C'est le DOM et le sujet du prochain cours!

Aspects négatifs

Qu’il faut connaître, qu’on devrait éviter

Variables globales

- **Variables globales!!!!**
  - Solution?
    - Créer un objet global et y sauvegarder les variables de l'application

```
var blogApp = {};
blogApp.background = "blue";
...
```

Aspects négatifs du langage

- **Variables globales!!!!**
  - Solution?
    - Utiliser les fermetures

Opérateurs de comparaison

- « **Abstract equality** »
  - `s1 == s2` si les types ne sont pas les mêmes, JavaScript fait une conversion, dont le résultat pourrait ne pas être le résultat souhaité
- « **Strict equality** »
  `v1 === v2`, `v1 !== v2` fait la comparaison strict (compare les références, identité)

Mais aussi

- Absence d'environnement lexical associé aux blocs d'énoncés
- « Correction d'erreurs? automatique »

```
return
{
   status: true
};
```

```
return {
   status: true
};
```

Mais encore

- **typeof null** retourne **object**
- Duplication de fonctionalités (fonction, new...)

Fermetures

« Closures »

Fermeture

- une **fermeture** capture des variables de son environnement lexical, ces dernières ne sont pas des variables globales

```
function makeAdder( op ) {
  return function( value ) {
    return op + value;
  };
}
```

```
js> var add1 = makeAdder( 1 );
js> var add5 = makeAdder( 5 );
js> add1( 1 );
2
js> add5( 1 );
6
```

Fermeture

- une **fermeture** capture des variables de son environnement lexical, ces dernières ne sont pas des variables globales

```
function makeFunc() {
  var name = "bar";
  function getName() {
    return name;
  }
  return getName();
}
```

```
js> name = "autre chose";
js> var myFunc = makeFunc();
js> myFunc();
bar
```

```
function makePair( f, s ) {
  var first = f;
  var second = s;
  var that = {};

  that.getFirst = function() {
    return first;
  }
  that.setFirst = function( f ) {
    first = f;
  }
  that.getSecond = function() {
    return second;
  }
  that.setSecond = function( s ) {
    second = s;
  }
  return that;
}
```

```
js> a = newPair( "Marcel", "Turcotte" );
[object Object]

js> b = newPair( "size", 800 );
[object Object]

js> a.getFirst();
Marcel

js> b.getFirst();
size

js> b.setFirst( "width" );

js> b.getFirst();
width

js> for ( propr in b ) { print( propr ) }
getFirst
setFirst
getSecond
setSecond
```

- Dans l'exemple de la page précédente, l'objet ne possède que des propriétés dont les valeurs sont des fonctions

- Les variables locales de la fonction makePair ont été capturées (la fermeture) et sont maintenant partagées par les méthodes **getFirst**, **setFirst**, **getSecond** et **setSecond**

- Ces variables ne sont pas des propriétés de l'objet et ne sont accessibles qu'à **getFirst** (et autres méthodes de l'objet)

- La capture de la **fermeture** de la méthode **makePair** réalise les objectifs de l'**encapsulation de données**
- On peut soi-même créer des objets dont les « attributs » sont vraiment immuables
- Cette implémentation court-circuite les mécanismes de sérialisation (introspection)

Fonctions

Deuxième partie

Appels de fonction

- **expr ()**
  où expression retourne une «fonction» (un objet dont le prototype est **Function.prototype**)

Function.prototype

- Une fonction est un objet dont le prototype est **Function.prototype** (qui est lui-même lié à **Object.prototype**)

Appels de fonction

- Prends **quatre formes** :
  - appel de méthode
  - appel de de fonction
  - appel de constructeur
  - appel à l'aide de **apply**
- **this** et **arguments** seront initialisées différemment

Appel de fonction

```
var add = function( x, y ) {
   return x + y
};
var sum = add( 3, 4 );
```

- Lors de l'invocation d'une fonction **this** est lié à l'objet global

Appel d'un constructeur

```
function Point() {
    this.x = 0;
    this.y = 0;
    this.getX = function() { return this.x };
    this.getY = function() { return this.y };
};

var p1 = new Point();
```

- Lors d'un appel à un constructeur
  - Un nouvel objet est créé ayant un lien implicite (caché) à la propriété **prototype** du constructeur (c'.-à-d. une fonction)
  - **this** est lié à l'objet nouvellement créé

Appel de méthode

```
function Point() {
    this.x = 0;
    this.y = 0;
    this.getX = function() { return this.x };
    this.getY = function() { return this.y };
};
```

- Lors de l'invocation d'une méthode **this** est lié à l'objet

```
var p1 = new Point();
...
x = p1.getX();
```

Appel à l'aide de apply

```
var somme = function() {
  var i, somme = 0;
  for ( i=0; i<arguments.length; i += 1 ) {
     somme += arguments[ i ];
  }
  return somme;
};

var ys = [ 1, 2, 3, 4, 5 ];

somme.apply( null, ys );

// retourne 15
```

- La méthode apply possède deux paramètre, le premier permet la liaison de **this**

Appel à l'aide de apply

```
function getName() {
   return this.name;
}
```

- La méthode apply possède deux paramètre, le premier permet la liaison de **this**

```
var person = { name: "Marcel", height: 178 };
var domain = { name: "uottawa.ca", mask: 255, owner: "U. of Ottawa" };


getName.apply( person );
getName.apply( domain );
```

Mais encore...

Ce qui n'as été présenté

- Expressions régulières
- JSON

Épilogue

Revue des concepts

- Les **valeurs ont des types** et non les variables
- Les **objets ont des propriétés** (qui ont des valeurs, qui elles ont des types)
- **Objets littérals** facilite la création d'objets
- Appels de fonctions par valeur

Revue des concepts

- **Programmation orientée prototype**
- Le mécanisme de **fermeture** permet l'**encapsulation de données** en JavaScript (équivalent de variables privées)
- JavaScript est un **langage fonctionnel** et **objet**, sans classes

Ressources

- Standard ECMA-262 ECMAScript Language Specification – 3rd edition (December 1999) [http://www.ecma-international.org/publications/standards/Ecma-262.htm] 2007
- Core JavaScript 1.5 Reference [http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference] 2007

Ressources (suite)

- ECMAScript [http://www.ecmascript-lang.org/] 2007
- Rhino – JavaScript for Java [http://www.mozilla.org/rhino] 2007
- Venkman JavaScript Debugger project page [http://www.mozilla.org/projects/venkman/] 2007
- JavaScript MDC [http://developer.mozilla.org/en/docs/JavaScript] visité le 26-janvier-2008

Ressources (suite)

- « **Closure Compiler** » de **Google** est un outil pour l'optimisation, mais surtout la compression de code JavaScript

http://code.google.com/closure/compiler

Ressources (suite)

- Doublas Crockford (2008) JavaScript: The Good Parts. O'Reilly.