

JavaScript



Thierry Chatel

MethoTIC Conseil



*Consultant indépendant
et formateur AngularJS*



tchatel@methotic.com



@ThierryChatel



+ThierryChatel



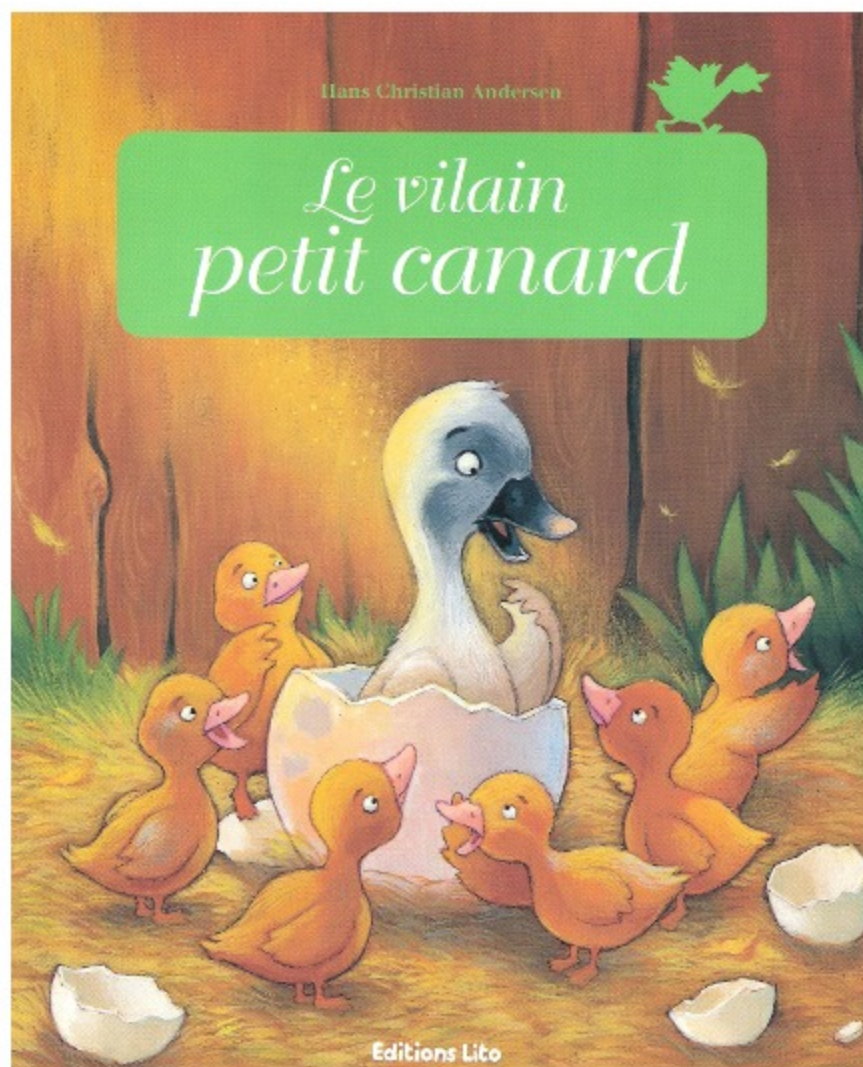
tchatel

slides et vidéo (version longue) :

<http://tinyurl.com/langage-javascript>

Hans Christian Andersen

*Le vilain
petit canard*



Editions Lito

Langage JavaScript

- orienté objet... sans classes
- non typé
- fonctionnel

Quel JavaScript ?

- ECMAScript 5 / 5.1 : décembre 2009 / juin 2011
 - mode strict
 - getters et setters
 - JSON
 - etc.

Quel JavaScript ?

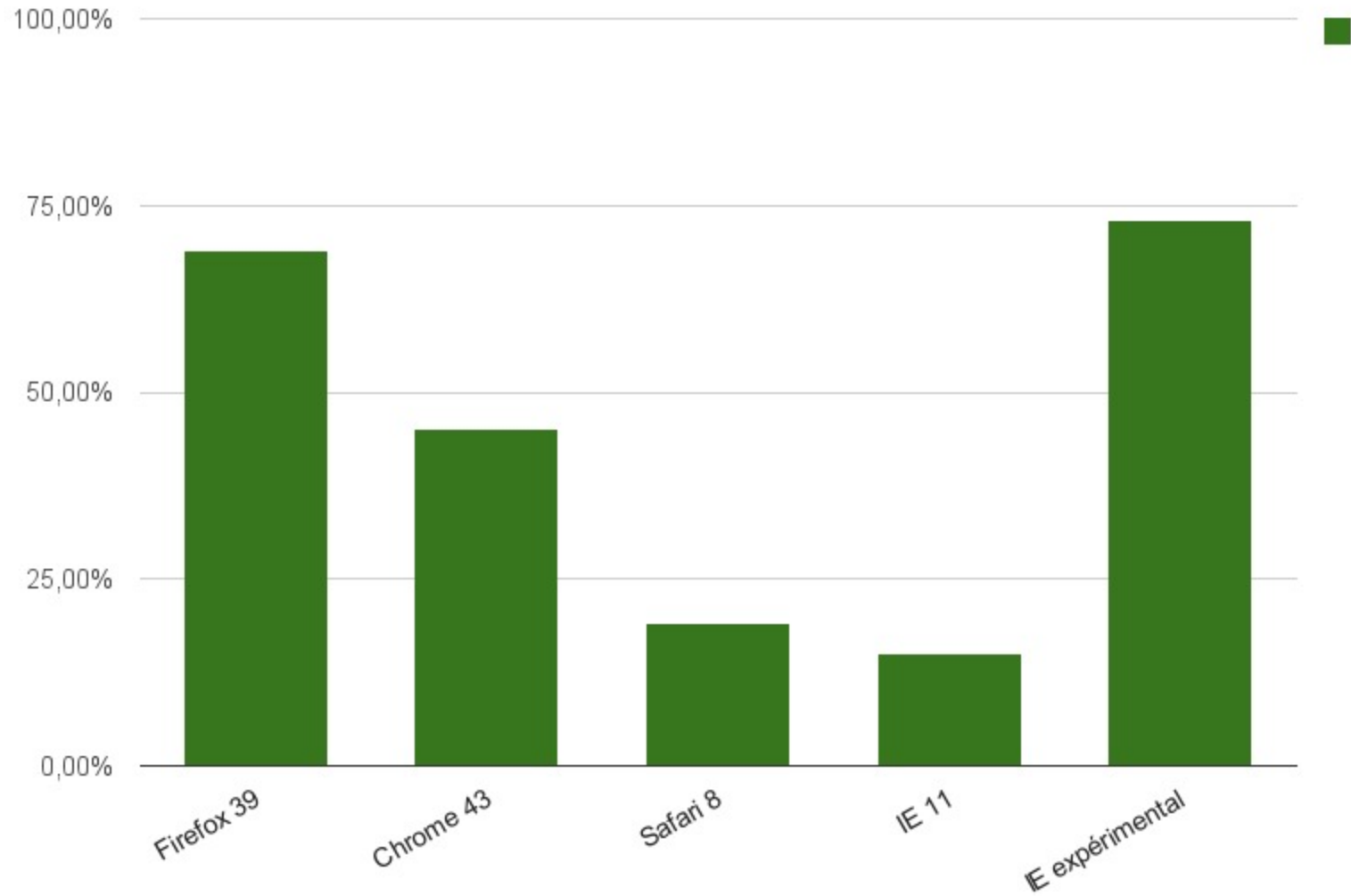
- ECMAScript 6 / 2015 : mi 2015

- classes
- modules
- promesses
- etc.

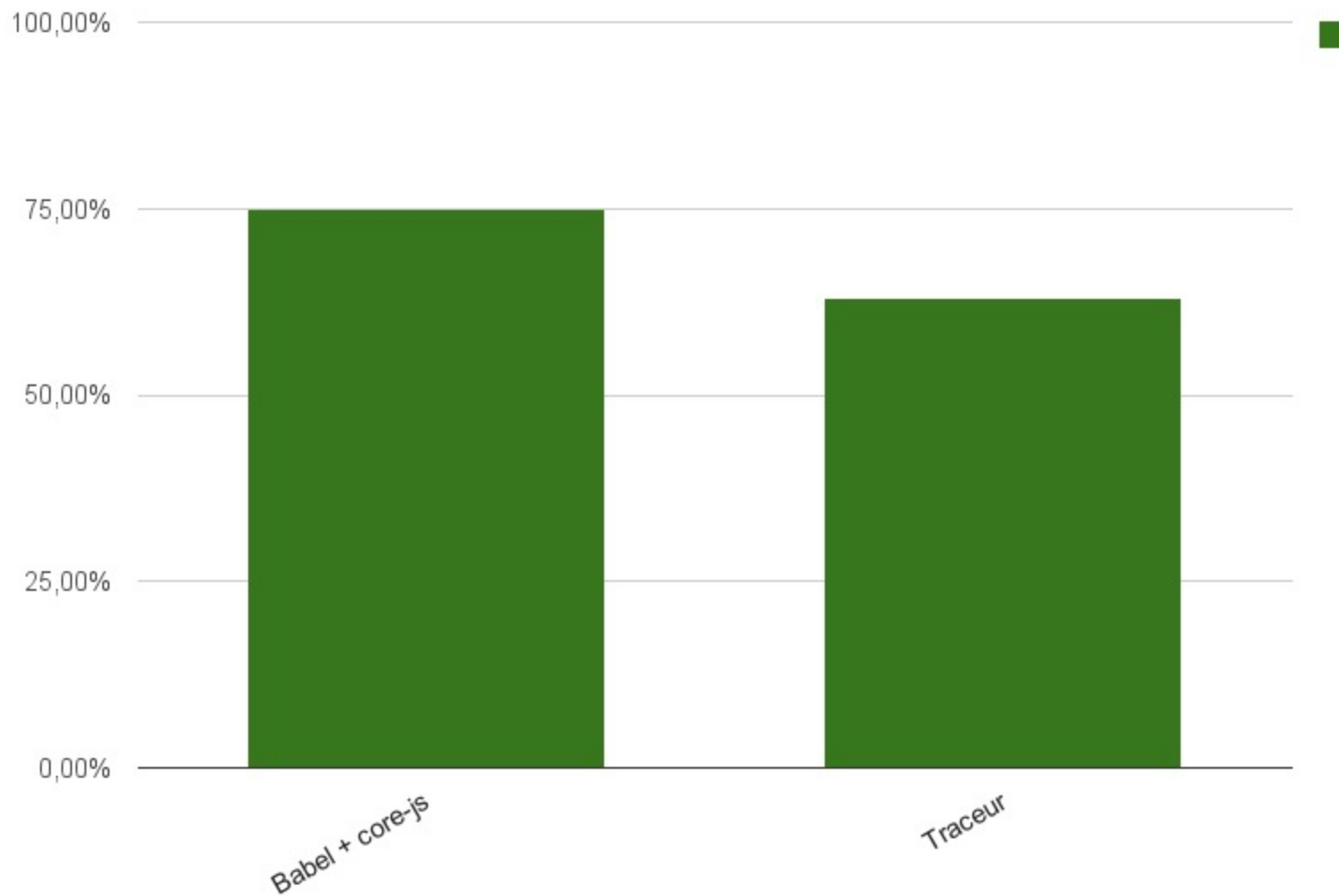
- *Support ES6 :*

<http://kangax.github.io/compat-table/es6/>

Fonctionnalités ES6 supportées par les navigateurs



ES6 : compilers / polyfills / transpilers



la syntaxe

Syntaxe JavaScript

```
"use strict";
function multiplicative() {
  var left = unary();
  var token;
  while ((token = expect('*', '/', '%'))) {
    left = binaryFn(left, token.fn, unary());
  }
  return left;
}
```

- mode strict : **"use strict"**
 - en tout début de fichier, ou au début d'une fonction

Mode strict

- signale davantage d'erreurs (*fail-fast*)
 - ex : variable non déclarée
- ne transforme plus **this** en objet
 - plus d'objet global **window** à la place de **undefined**
- impose
 - l'unicité des propriétés dans un objet littéral
 - l'unicité des noms d'arguments dans une fonction
 - les déclarations de fonctions au plus haut niveau d'un script ou d'une fonction

Syntaxe JavaScript

```
"use strict";
function multiplicative() {
  var left = unary();
  var token;
  while ((token = expect('*', '/', '%'))) {
    left = binaryFn(left, token.fn, unary());
  }
  return left;
}
```

- **var** pour déclarer une variable (*non typée*)

Syntaxe JavaScript

```
"use strict";
function multiplicative() {
  var left = unary();
  var token;
  while ((token = expect('*', '/', '%'))) {
    left = binaryFn(left, token.fn, unary());
  }
  return left;
}
```

- point-virgule en fin d'instruction
 - pouvant être omis (*déconseillé !*)

Semicolon insertion

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

- JavaScript rajoute automatiquement les ; manquants
 - et ce n'est pas désactivable !

Semicolon insertion

```
function f(x) {  
  return  
  [  
    1,  
    "abc",  
    true  
  ];  
}
```

- que renvoie la fonction f ?

Semicolon insertion

- Règles d'insertion automatique des ;
 - en fin de ligne, de fichier, ou avant une }
 - seulement si le token suivant crée une erreur de syntaxe
 - jamais dans l'entête d'une boucle *for*
 - jamais quand le ; crée une instruction vide
 - insertion systématique là où une fin de ligne est interdite (*voir liste au dos*)

Semicolon insertion

- Règles d'insertion automatique des ;
 - insertion systématique là où une fin de ligne est interdite :
 - PostfixExpression :
 - **LeftHandSideExpression** [no LineTerminator here] ++/--
 - ContinueStatement & BreakStatement :
 - **continue/break** [no LineTerminator here] **Identifier** ;
 - ReturnStatement :
 - **return** [no LineTerminator here] **Expression** ;
 - ThrowStatement :
 - **throw** [no LineTerminator here] **Expression** ;

Semicolon insertion

- Ouvrir les accolades et les crochets en fin de ligne

```
return {  
    s: 'azerty',  
    n: 123  
};
```

Scope

- Portée des variables en JavaScript : “*function scope*”
 - une variable est déclarée pour toute la fonction qui la contient
 - même avant la ligne où se trouve le mot-clef *var*
- contrairement au “*block scope*” de Java : {...}

Function scope

```
function f(array) {  
  // i, element, label sont déjà déclarées ici  
  for (var i = 0 ; i < array.length ; i++) {  
    var element = array[i];  
    if (element.max > 0) {  
      var label = element.label;  
      // ...  
    }  
  }  
}
```

Function scope

- Une variable déclarée hors de toute fonction devient une propriété du scope global
 - dans un navigateur web, le scope global est accessible sous le nom de **window**

les types

5 types primitifs

- boolean, number, string, undefined, null
 - *undefined* : si pas de valeur affectée
 - *null* : affecté explicitement, pour signifier l'absence d'objet
 - un seul type numérique, sur 8 octets
 - pas de type caractère

primitif / objet

- Type primitif
 - pas de référence
 - passage par valeur (*copie de la valeur*)

- Objet
 - passage par référence

Autoboxing

- Les booléens, nombres et chaînes de caractères sont :
 - immuables
 - convertis à la volée en objets (**Boolean**, **Number**, **String**) quand c'est nécessaire

```
var n = 5;  
n.toString();  
  
var s = "hello";  
s.length;
```

Autoboxing

- Ne pas utiliser explicitement les types objets **Boolean**, **Number** et **String**

```
var a = new String('AZERTY');  
var b = new String('AZERTY');  
a == 'AZERTY'; // true  
a === 'AZERTY'; // false  
a == b; // false  
a === b; // false  
typeof 'AZERTY'; // "string"  
typeof a; // "object"
```

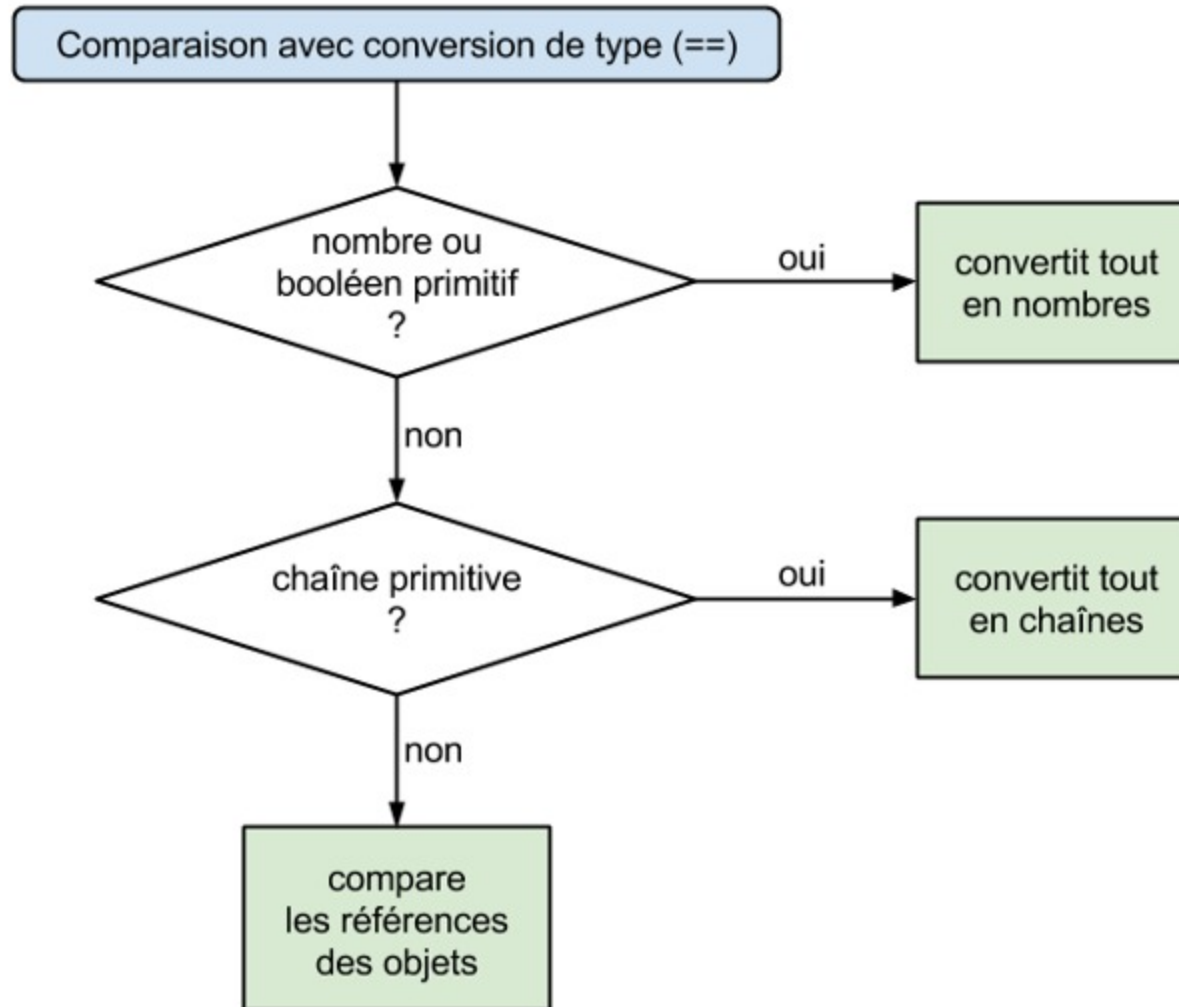
Comparaisons

- **===** Comparaison stricte (*sans conversion automatique*)
 - **true** pour deux données primitives du même type et de même valeur, c'est-à-dire :
 - **true** pour deux booléens (primitifs) égaux
 - **true** pour deux nombres (primitifs) égaux
 - **true** pour deux chaînes de caractères (primitives) égales
 - **true** pour deux références au même objet
 - **false** dans tous les autres cas

Comparaisons

- **==** Comparaison avec conversion automatique de type
 - s'il y a un opérande primitif
 - **null == undefined**
 - si un opérande **number** ou un **boolean** => convertit tout en **number**
 - si un opérande **string** => convertit tout en **string**
 - pour deux objets : compare les références

Comparaisons



Comparaisons

```
0 == false;  
1 == true;  
"1" == true;  
2 == "2";  
1 == new Number(1);  
new Number(1) != new Number(1);  
"" == 0  
false == ""  
false == []  
false != {}  
[] == ""  
["2"] == 2
```

Comparaisons

```
// string -> number  
"" == 0
```

```
// Array -> number  
["2"] == 2  
[""] == 0  
[""] == false  
["2"] != [2]
```

```
// Array -> string  
["2", 3] == "2,3"
```


Comparaisons

```
// Non transitif  
"1" == true;  
"01" == true;  
"01" != "1";
```

les objets

Les objets

- Les objets JavaScript sont des *maps* (clefs / valeurs)
 - clef : chaîne de caractères = nom de la propriété
 - valeur quelconque (non typé)

Les objets

- Les objets JavaScript sont des *maps* (clefs / valeurs)
 - clef : chaîne de caractères = nom de la propriété
 - valeur quelconque (non typé)
- Pas de notion de visibilité : toutes les propriétés sont publiques

Les objets

- Les objets JavaScript sont des *maps* (clefs / valeurs)
 - clef : chaîne de caractères = nom de la propriété
 - valeur quelconque (non typé)
- Pas de notion de visibilité : toutes les propriétés sont publiques
- 2 façons d'accéder à la valeur d'une propriété
 - `obj['clef']`
 - `obj.clef`

Les objets

- Notation entre crochets :

- possible avec un nom de propriété dynamique
- n'importe quelle chaîne autorisée comme nom de propriété

```
obj['a.b']  
obj['']
```

- Notation avec un point :

- possible pour un nom de propriété en dur, qui est un **identifiant valide**

```
obj.a.b // N'a pas le même sens !
```

Les objets

- Créer un objet, avec `{...}`

```
var DATE_FORMATS = {  
  yyyy: dateGetter('FullYear', 4),  
  yy: dateGetter('FullYear', 2, 0, true),  
  MM: dateGetter('Month', 2, 1),  
  dd: dateGetter('Date', 2),  
  HH: dateGetter('Hours', 2),  
  hh: dateGetter('Hours', 2, -12),  
  mm: dateGetter('Minutes', 2),  
  ss: dateGetter('Seconds', 2),  
  Z: timeZoneGetter  
};
```

Les objets

- Créer des objets imbriqués

```
var book = {  
  title: "Javascript: The Good Parts",  
  author: "Douglas Crockford",  
  publisher: "O'Reilly",  
  details: {  
    "ISBN-10": "0596517742",  
    "ISBN-13": "978-0596517748",  
    pages: 176  
  }  
};
```


Les objets

- Ajouter ou modifier une propriété

```
angular.element = jqLite;
```

```
dst[key] = src[key];
```

- Supprimer une propriété

```
delete params[key];
```

```
delete event.stopPropagation;
```

EXAMPLE

```
'use strict';

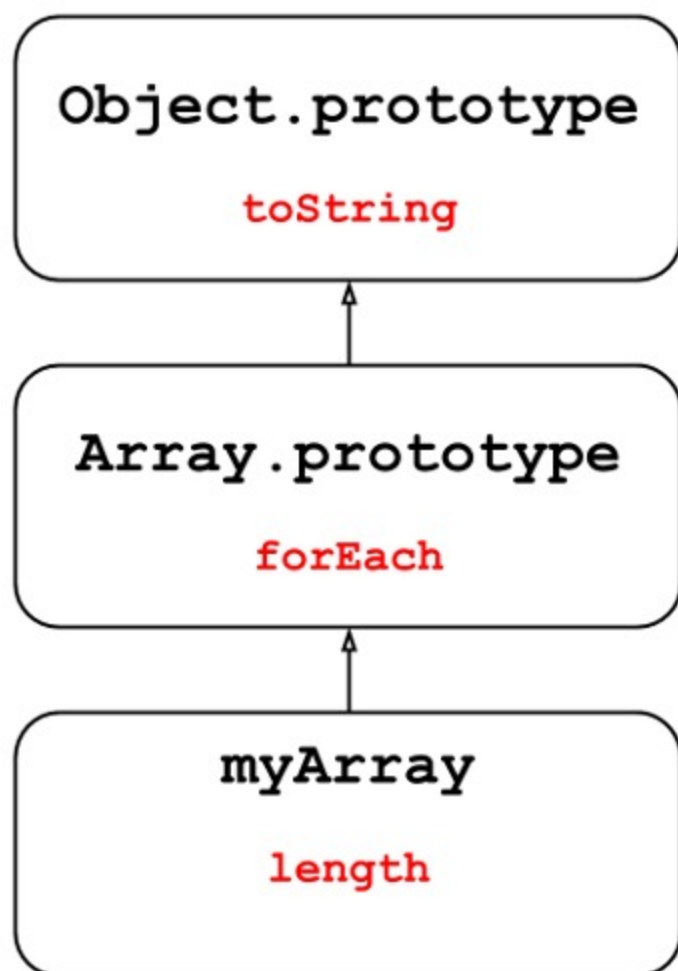
var me = {
  firstName: "Thierry",
  lastName: "Chatel",
  fullName: function () {
    return this.firstName + " "
      + this.lastName;
  }
};

console.log(me.fullName());
```

Chaîne des prototypes

- Recherche d'une propriété (*get*)
 - dans l'objet courant
 - puis dans son prototype
 - puis dans le prototype de son prototype
 - puis dans le prototype du prototype de son prototype
 - ...

Chaîne des prototypes



lecture : héritage

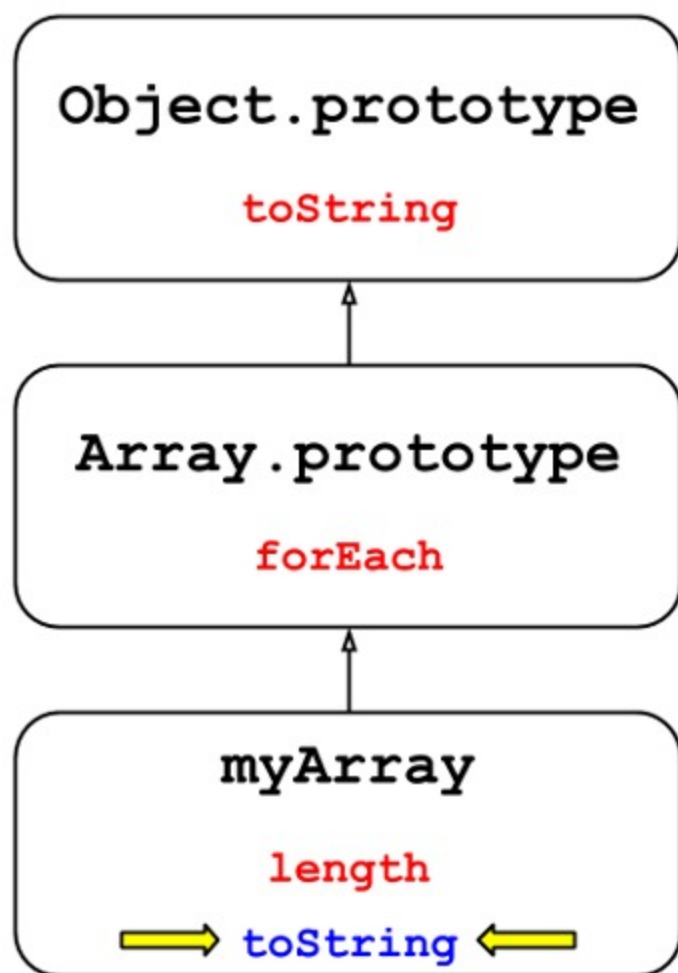
myArray

length
forEach
toString

Chaîne des prototypes

- Affectation d'une valeur à une propriété (*set*)
 - modifie ou crée la propriété **toujours dans l'objet courant**
 - même si c'était une propriété héritée d'un prototype

Chaîne des prototypes



écriture : en local

`myArray.toString = ...`

Chaîne des prototypes

- Accéder au prototype d'un objet :
 - ES5 : `Object.getPrototypeOf(obj)`
 - pas standard : `obj.__proto__`
- Racine de la chaîne des prototypes :
`Object.prototype`

```
Object.getPrototypeOf(Object.prototype) ; //null
```

- Créer un objet avec un certain prototype :

```
var obj = Object.create(proto) ;
```


les tableaux

Les tableaux

- Les tableaux sont des objets JavaScript, initialisés avec `[...]`
 - comme pour tous les objets, les noms de propriétés sont des chaînes de caractères

```
var emptyArray = [];
```

```
var a = ['this', 'is', 'an', 'array'];  
a[1]; // "is"
```

Les tableaux

- Certaines propriétés sont considérées comme des **index numériques**
 - **représentation texte canonique d'un entier positif**
 - index : "0", "1", "2", "15"
 - pas index : "-1", "3.14", "01"

```
var a = ['this', 'is', 'an', 'array'];  
Object.keys(a); // ["0", "1", "2", "3"]  
a[1]; // "is"  
a['1']; // "is"
```

Les tableaux

- Propriété *length* > plus grand index
 - ajout d'un index trop grand : *length* augmente
 - *length* ne diminue jamais automatiquement
 - modification de *length* : les index trop grands sont supprimés

```
a[5] = 'new';  
a.length; // 6  
a.length = 2;  
a; // ["this", "is"]  
a.length = 0;  
a; // []
```

Les tableaux

- Méthodes modifiant le contenu du tableau
 - **array.push(element1, element2, ..., elementN)**
ajoute en fin de tableau
 - **array.pop()**
enlève et renvoie le **dernier** élément
 - **array.shift()**
enlève et renvoie le **premier** élément
 - **array.unshift(element1, ..., elementN)**
ajoute en début du tableau
 - **array.splice(index, howMany, element1, ...)**
supprime et/ou ajoute des éléments
 - **array.sort([compareFunction])**
trie les éléments à l'intérieur du tableau
 - **array.reverse()**
inverse l'ordre des éléments d'un tableau

Les tableaux

- Méthodes créant un nouveau tableau
 - **array.concat(value1, value2, ..., valueN)**
renvoie un nouveau tableau égal à la concaténation du tableau courant et des tableaux passés en paramètres
(les arguments qui ne sont pas des tableaux sont ajoutés comme des éléments)
 - **array.slice(begin[, end])**
renvoie une copie d'une portion du tableau courant

Les tableaux

- Méthodes ajoutées dans ECMAScript 5
 - **array.indexOf(searchElement[, fromIndex])**
renvoie le premier index où l'élément est trouvé
 - **array.lastIndexOf(searchElement[, fromIndex])**
renvoie le dernier index où l'élément est trouvé
 - **array.forEach(callback[, thisArg])**
exécute une fonction callback pour chaque élément du tableau

Les tableaux

- Méthodes ajoutées dans ECMAScript 5 (*suite*)
 - **array.every(callback[, thisObject])**
renvoie *true* si tous les éléments du tableau passent le test de la fonction callback
 - **array.some(callback[, thisObject])**
renvoie *true* si au moins un des éléments du tableau passe le test de la fonction callback
 - **array.filter(callback[, thisObject])**
renvoie un tableau contenant les éléments qui passent le test de la fonction callback
 - **array.map(callback[, thisArg])**
renvoie un tableau avec le résultat de la fonction callback pour chacun des éléments du tableau de départ
 - **array.reduce(callback[, initialValue])**
renvoie la valeur calculée en appliquant une fonction avec un accumulateur à chaque élément du tableau, de gauche à droite

EXAMPLE

```
'use strict';

var numbers = [ 17, 35, 46, 73, 20, 52, 67 ];

numbers.maxFor = function () {
  var max;
  for (var i = 0 ; i < this.length ; i++) {
    if (max === undefined || this[i] > max) {
      max = this[i];
    }
  }
  return max;
};

console.log(numbers.maxFor());
```

```
'use strict';

var numbers = [ 17, 35, 46, 73, 20, 52, 67 ];

numbers.maxForEach = function () {
  var max;
  this.forEach(function (item) {
    if (max === undefined || item > max) {
      max = item;
    }
  });
  return max;
};

console.log(numbers.maxForEach());
```

```
'use strict';

var numbers = [ 17, 35, 46, 73, 20, 52, 67 ];

numbers.maxReduce = function () {
  return this.reduce(function (max, item) {
    return item > max ? item : max;
  });
};

console.log(numbers.maxReduce());
```

définition de fonctions

Définition de fonctions

- *FunctionDeclaration* :

```
function Identifier(FormalParameterList<opt>) {  
    FunctionBody  
}
```

- *FunctionExpression* :

```
function Identifier<opt>(FormalParameterList<opt>) {  
    FunctionBody  
}
```

Définition de fonctions

- **Déclaration**

```
function f (arg1, arg2) {  
    ...  
}
```

- mot-clef ***function*** en début d'instruction
- nom est obligatoire
- pas de valeur

Définition de fonctions

- Expression

```
return function f(arg1, arg2) {  
    ...  
};
```

- mot-clef ***function*** dans une expression (*pas en début d'instruction*)
- nom facultatif (*utile pour une fonction récursive*)
- valeur : l'objet fonction

Définition de fonctions

```
function f1() { // Declaration
    return function () { // Expression
    };
}

setTimeout(function () { // Expression
    // ...
}, 1000);

var f3 = function () { // Expression
    function f33() { // Declaration
    }
};
```

Fonctions internes

- **déclaration *function***
 - définies dès le début de la fonction qui les contient
 - ou dès le début du fichier si hors de toute fonction
- **expression *function***
 - définies seulement après l'évaluation de l'expression

Fonctions internes

```
// f est utilisable ici

function f() {
  // f1 est utilisable ici
  // mais pas f2

  function f1() {
    // ...
  }
  var f2 = function () {
    // ...
  };
}
```

First class objects

- Les fonctions sont de vrais objets JavaScript
- On peut :
 - mettre une fonction dans une variable
 - mettre une fonction en propriété d'un objet
 - mettre une fonction dans un tableau
 - passer une fonction comme paramètre d'une autre fonction
 - renvoyer une fonction

First class objects

- On peut ajouter des propriétés à une fonction.

```
function myController($loc, $log) {  
    // ...  
}  
// which services to inject ?  
myController.$inject =  
    ['$location', '$log'];
```

EXAMPLE

- sans cache

```
'use strict';  
  
function fact(n) {  
    return n > 0 ? n * fact(n - 1) : 1;  
}  
  
console.log(fact(5));
```

- avec cache

```
'use strict';

function fact(n) {
  if (!fact.cache[n]) {
    console.log("calcul", n);
    fact.cache[n] =
      n > 0 ? n * fact(n - 1) : 1;
  }
  return fact.cache[n];
}

fact.cache = {};

console.log(fact(5));
console.log(fact(7));
```


- avec cache (*variante*)

```
'use strict';

function fact(n) {
  if (!fact[n]) {
    console.log("calcul", n);
    fact[n] = n > 0 ? n * fact(n - 1) : 1;
  }
  return fact[n];
}

console.log(fact(5));
console.log(fact(7));
```

appel de fonctions

4 façons d'invoquer une fonction

- appel de fonction classique
- appelée comme une méthode
- appelée comme un constructeur
- avec **call** ou **apply**

1ère façon : appel classique

```
var data = transformData(  
    response.data,  
    response.headers,  
    respTransformFn  
);
```

1ère façon : appel classique

- Paramètres

- pas assez de valeurs : *undefined*
- trop de valeurs : ignorées, mais accessibles dans l'objet *arguments*
- *arguments* est un objet qui ressemble à un tableau
 - **arguments.length**
 - **arguments[0]**
 - **arguments[1]**
 - ...

1ère façon : appel classique

- **this** est égal
 - à **undefined** en mode strict
 - au scope global (*window*) en mode normal !?!

EXAMPLE

```
'use strict';

function sum() {
    var total = 0;
    for (var i = 0 ;
         i < arguments.length ;
         i++) {
        total += arguments[i];
    }
    return total;
}

console.log(sum(1, 2, 3, 4, 5));
```


2ème façon : comme méthode

- la fonction est une propriété d'un objet
- appel faisant référence explicitement à la propriété de l'objet

```
filtered.push(value);
```

```
filtered['push'](value);
```

- dans la fonction, **this** est égal à l'objet qui précède la propriété

2ème façon : comme méthode

- Attention : la valeur de **this** est perdue dans les fonctions internes

```
controller: function($element, $scope, $attrs) {  
    var self = this;  
    $scope.$on('$destroy', function() {  
        self.renderUnknownOption = noop;  
    });  
},
```

- sauf dans les *arrow functions* de ES6/2015 (=>)

```
$scope.$on('$destroy',  
    () => {this.renderUnknownOption = noop;});
```

EXAMPLE

- pas bon :

```
'use strict';

var logger = {
  logThis: function () {
    console.log(this);
  }
};

setTimeout(logger.logThis, 2000);
```

- ok : dans une fonction anonyme

```
'use strict';

var logger = {
  logThis: function () {
    console.log(this);
  }
};

setTimeout(function () {
  logger.logThis();
}, 2000);
```

3ème façon : comme constructeur

```
var location = new LocationUrl(  
    convertToHtml5Url (initUrl,  
                        basePath,  
                        hashPrefix),  
    pathPrefix,  
    appBaseUrl  
);
```

- mot-clef **new**
 - utilisable avec toute fonction JS
 - convention :
le nom de la fonction commence par une majuscule

3ème façon : comme constructeur

- JS crée un nouvel objet **location**
 - avec pour prototype **LocationUrl.prototype**
 - JS ajoute une propriété **prototype** à toute fonction
 - vérifiant **location instanceof LocationUrl**
 - **instanceof** vérifie si **LocationUrl.prototype** est dans la chaîne des prototypes de l'objet
 - vérifiant **location.constructor === LocationUrl**
 - parce que la propriété **constructor** est définie dans son prototype **LocationUrl.prototype**

3ème façon : comme constructeur

- ... puis appelle la fonction utilisée comme constructeur
 - où **this** référence le nouvel objet

```
function LocationUrl(url,
                    pathPrefix,
                    appBaseUrl) {
  this.parse = function(newAbsoluteUrl) {
    // ...
  };
  this.compose = function() {
    // ...
  };
  this.parse(url);
}
```


3ème façon : comme constructeur

- ... et le constructeur ne renvoie rien.
 - s'il renvoie un type primitif, son retour est ignoré
 - s'il renvoie un objet, son retour remplace l'objet créé par **new**
 - *fortement déconseillé !*

3ème façon : comme constructeur

- Ajout de fonctions au prototype
 - fonctions partagées par tous les objets liés à ce prototype
 - même ceux créés avant l'ajout de la fonction au prototype

```
LocationUrl.prototype.f =  
  function (arg1, arg2) {  
    // ...  
  };
```

3ème façon : comme constructeur

- Risque d'oublier le **new**
 - erreur en mode strict, car **this** vaut **undefined**

EXAMPLE

```
'use strict';

function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

var p1 = new Person("Thierry", "Chatel");

Person.prototype.fullName = function () {
    return this.firstName + ' '
        + this.lastName;
};

var p2 = new Person("Jerry", "Khan");

console.log(p1.fullName());
console.log(p2.fullName());
```

4ème façon : call & apply

- **call** : appel d'une fonction, en fournissant
 - la valeur de **this**
 - les paramètres un à un

```
fn.call(thisArg, arg1, arg2, arg3);
```

4ème façon : call & apply

- **apply** : appel d'une fonction, en fournissant
 - la valeur de **this**
 - un tableau de paramètres

```
fn.apply(thisArg, [arg1, arg2, arg3]);
```

4ème façon : call & apply

- Exemple : la méthode `each()` de jQuery

```
// Utilisation : this est l'élément HTML courant
jQuery('a').each(function(index, value) {
    var ihref = $(this).attr('href');
    if (ihref.indexOf("http") >= 0) {
        console.log(ihref+'<br/>');
    }
});

// dans le code du each :
for ( ; i < length; i++ ) {
    value = callback.call( obj[ i ], i, obj[ i ] );
}
```


ES6/2015 : class

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  toString() {  
    return '(' + this.x + ', '  
           + this.y + ')';  
  }  
}
```

- une fonction constructeur
- des méthodes sur le prototype associé

ES6/2015 : class

- équivalent à (ES5) :

```
function Point(x, y) {  
  this.x = x;  
  this.y = y;  
}  
Point.prototype.toString = function () {  
  return '(' + this.x + ', ' + this.y + ')';  
};
```

ES6/2015 : class

```
class Rectangle {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }
  static area(width, height) {
    return width * height;
  }
  area() {
    return Rectangle.area(this.width,
                           this.height);
  }
}
```

- **static** : propriété de la fonction constructeur

ES6/2015 : class

```
class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y);
    this.color = color;
  }
  toString() {
    return super.toString()
      + ' in ' + this.color;
  }
}
```

- *peu utile en JS !*

ES6/2015 : class

- équivalent à (ES5) :

```
function ColorPoint(x, y, color) {
    Point.call(this, x, y);
    this.color = color;
}
ColorPoint.prototype = new Point();
ColorPoint.prototype.constructor = ColorPoint;

ColorPoint.prototype.toString = function () {
    return Point.prototype.toString.call(this)
        + ' in ' + this.color;
};
```

Closure

- Toute fonction JavaScript a accès aux données du scope parent dans lequel elle est définie.

```
controller: function($element,  
                    $scope,  
                    $attrs) {  
    var self = this;  
    $scope.$on('$destroy', function() {  
        self.renderUnknownOption = noop;  
    <});  
},
```

- **self** est accessible dans la fonction interne car c'est une variable du scope parent

Closure

- **Fonctionnement**
 - la fonction garde une référence vers le scope parent (variables, paramètres et fonctions)
 - les données sont celles présentes dans le scope **au moment de l'exécution** de la fonction
 - et pas au moment de sa définition (*aucune copie de l'état du scope*)

Closure

- Conséquences
 - une fonction globale a accès aux données du scope global
 - une fonction interne a accès aux données de son scope parent
 - = celles de la fonction dans laquelle elle est définie
 - ... qui a elle-même accès aux données de son scope parent
 - ... et ce jusqu'au scope global

Closure

```
'use strict';
var v0 = 0;
function f1(arg1) {
  var v1 = 1;
  return function f2(arg2) {
    var v2 = 2;
    return function f3(arg3) {
      var v3 = 3;
      console.log(v0, v1, v2, v3);
      console.log(arg1, arg2, arg3);
    }
  }
}
f1('aaa')('bbb')('ccc');
```

Closure

- Attention aux *memory leaks* !
- JS purge le scope à la sortie de la fonction parente
 - conserve les propriétés appelées dans les *closures*
 - y compris les *closures* qui ne sont plus référencées !
- Solution : annuler les propriétés ne devant pas être conservées

EXAMPLE

- pas bon :

```
'use strict';

function count() {
  for (var i = 1; i <= 10 ; i++) {
    setTimeout(function () {
      console.log(i);
    }, i * 1000);
  }
}

count();
```

- ok : avec un appel de fonction

```
'use strict';

function count() {
  function log(i) {
    setTimeout(function () {
      console.log(i);
    }, i * 1000);
  }
  for (var i = 1; i <= 10 ; i++) {
    log(i);
  }
}

count();
```

- ok : avec un anonymous wrapper

```
'use strict';

function count() {
  for (var i = 1; i <= 10 ; i++) {
    (function log(i) {
      setTimeout(function () {
        console.log(i);
      }, i * 1000);
    })(i);
  }
}

count();
```

Anonymous wrapper

- Fonction anonyme exécutée immédiatement
 - entre parenthèses, pour que ce soit une expression
 - avec ou sans arguments

```
(function (arg1, arg2) {  
    // ...  
})(a1, a2);
```

Anonymous wrapper

- Fonction anonyme exécutée immédiatement
 - entre parenthèses, pour que ce soit une expression
 - avec ou sans arguments

```
(function (arg1, arg2) {  
    // ...  
})(a1, a2);
```


Anonymous wrapper

- Fonction anonyme exécutée immédiatement
 - Alternative avec moins de parenthèses

```
!function (arg1, arg2) {  
    // ...  
} (a1, a2);
```

Anonymous wrapper

```
(function () {  
    // ...  
})();
```

- Usage : créer un scope contenant les variables et fonctions
 - ne pas polluer le scope global
 - encapsuler le contenu de chaque fichier dans un anonymous wrapper

Pattern “Module”

- factory anonyme appelée immédiatement
 - variables locales = données & fonctions privées du module
 - renvoie un objet
 - avec des propriétés et méthodes publiques
 - qui ont accès aux données & fonctions privées

Pattern “Module”

```
var module = (function () {  
  var privateVariable = 1;  
  var privateMethod = function () {  
    // ...  
  };  
  return {  
    moduleProperty: 1,  
    moduleMethod: function () {  
      // ...  
      privateVariable = 2;  
      privateMethod();  
    }  
  };  
})();
```

EXAMPLE

```
// Module (= anonymous factory)
var c = (function () {
  var value = 0;
  return {
    get: function () {
      return value;
    },
    inc: function (i) {
      value += i || 1;
    }
  };
})();

console.log(c.get());
c.inc();
console.log(c.get());
c.inc(4);
console.log(c.get());
```

```
// Factory
function counterFactory(init) {
  var value = init;
  return {
    get: function () {
      return value;
    },
    inc: function (i) {
      value += i || 1;
    }
  };
}

var c1 = counterFactory(10),
    c2 = counterFactory(20);
console.log(c1.get(), c2.get());
c1.inc();
console.log(c1.get(), c2.get());
c2.inc(4);
console.log(c1.get(), c2.get());
```

```
// Constructor
```

```
function Counter(init) {  
    var value = init;  
    this.get = function () {  
        return value;  
    };  
    this.inc = function (i) {  
        value += i || 1;  
    };  
}  
  
var c1 = new Counter(10),  
    c2 = new Counter(20);  
console.log(c1.get(), c2.get());  
c1.inc();  
console.log(c1.get(), c2.get());  
c2.inc(4);  
console.log(c1.get(), c2.get());
```



```
// Constructor + prototype
```

```
function Counter(init) {  
    this._value = init;  
}
```

```
Counter.prototype.get = function () {  
    return this._value;  
};
```

```
Counter.prototype.inc = function (i) {  
    this._value += i || 1;  
};
```

```
var c1 = new Counter(10),  
    c2 = new Counter(20);  
console.log(c1.get(), c2.get());  
c1.inc();  
console.log(c1.get(), c2.get());  
c2.inc(4);  
console.log(c1.get(), c2.get());
```

JS patterns

Duck typing

“If it walks like a duck and quacks like a duck, it's a duck.”

- exemple

```
function logValue(it) {  
    var next = it.next();  
    console.log(next);  
}
```

Duck typing

```
function sum() {  
    var args =  
        Array.prototype.slice.call(arguments);  
    return args.reduce(function (sum, item) {  
        return sum + item;  
    }, 0);  
}
```

- *NOTE: The slice function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.*

Duck typing

- typage par l'usage, contrat implicite
- sans typage statique, pas besoin :
 - de classes
 - d'interfaces
 - d'héritage
- le prototype n'est qu'un moyen de partager du code

Mixins

- *extend*

```
function sum() {
  angular.extend(arguments, {
    map:    Array.prototype.map,
    reduce: Array.prototype.reduce
  });

  return arguments.reduce(
    function (sum, item) {
      return sum + item;
    },
    0
  );
}
```

Configuration object

```
$http({  
  method: 'GET',  
  url: 'https://angularjs.org/',  
  cache: true,  
  withCredentials: true  
});
```

```
// Beaucoup mieux que  
$http('GET', 'https://angularjs.org/',  
      undefined, undefined, undefined,  
      true, undefined, true);
```

Promesses

- Utiliser les *promesses* pour gérer l'asynchronisme.
- Disponibles dans de nombreux frameworks, et en ES6/2015.
- Promesse :
 - objet qui représente un résultat asynchrone
 - ne contient jamais le résultat
 - méthode **then** pour enregistrer :
 - un callback de succès
 - un callback d'erreur
 - on appelle **then ()** que la promesse soit en attente ou déjà résolue

slides et vidéo (version longue) :

<http://tinyurl.com/langage-javascript>