



# DÉVELOPPEMENT INFORMATIQUE LOGIQUE & PROGRAMMATION

LANGUAGE JAVASCRIPT

Y. DELVIGNE

CH. LAMBEAU



# JAVASCRIPT

## CONSIDÉRATIONS PRÉLIMINAIRES

## ORIGINES

JavaScript, originally called LiveScript, was developed by Brendan Eich at Netscape in 1995 and was shipped with Netscape Navigator 2.0 beta releases

JavaScript n'a rien à voir avec Java; le nom est un "coup marketing" pour profiter de la "vague Java" de l'époque ☺

## STANDARDISATION

Fin des années 1990 : "Browsers War" (Microsoft ⇔ Netscape) → incompatibilités

Standardisation : ECMAScript ("définition" du langage,

version actuelle ECMA-252 edition 5)

JavaScript = implémentation de ECMAScript dans un browser (Chrome, Firefox, ...)

## LANGUAGE

JavaScript is a full-featured, general-purpose,  
object-based, prototype-based  
interpreted programming language  
for adding active behavior to web pages

## LANGUAGE → RÈGLES GRAMMATICALES ET SYNTAXIQUES (1A)

- rédaction : texte brut en ASCII
  - pas de caractères accentués, p.ex.  
(sauf pour les "textes" bien entendu)
- le script (programme) est constitué de deux types de "mots"

- 1) les mots réservés du langage lui-même (en minuscules)

**break case catch continue debugger default delete  
do else false finally for function if in instanceof  
new null return switch this throw true try typeof  
var void while with**

**!!** le langage est "case sensitive" : les mots réservés doivent être écrits en minuscules (sinon erreur)

on rencontrera quelques mots réservés composés : ils doivent être obligatoirement rédigés en "camelCase" : **parseInt, parseFloat**

## LANGAGE → RÈGLES GRAMMATICALES ET SYNTAXIQUES (1B)

- le script (programme) est constitué de deux types de "mots"
  - 2) les mots créés par le programmeur, appelés identificateurs
    - ils doivent obligatoirement commencer par une lettre (minuscule) et ne peuvent contenir d'autres caractères que des lettres non accentuées (majuscules ou minuscules) ou des chiffres ou le caractère \_ (underscore)  
ex. : **quantite, prix, nb1, nb2**
    - sauf exception, ils sont rédigés en minuscules et au besoin utilisent la notation "camelCase"  
ex. : **prixUnit, nbEtu, ecartType, estBisseur**
    - le langage est "case sensitive" : les identificateurs doivent être écrits tels qu'ils ont été déclarés  
ex. : **prixUnit ≠ prixUNIT ≠ PrixUnit !!!**

## LANGAGE → RÈGLES GRAMMATICALES ET SYNTAXIQUES (2A)

- le script est constitué d'instructions (des "phrases", des "ordres")
- les instructions 'simples' tiennent en une seule ligne et se terminent (sauf exception) par un ;

```
ex. : var a, b, c;           // déclaration
      a = 12;                // affectation de valeur
      b = 2;                 // affectation de valeur
      c = a * b;              // affectation de résultat
                                d'expression
```

- des instructions plus 'complexes' comportent une séquence d'instructions (appelée aussi bloc)  
commençant par { et se terminant par } il n'y a pas de ; final

```
ex. : var a = 2, b = 12, c, d;
      if (a < b) {
        c = a * b;
        d = a / b;
      }
```

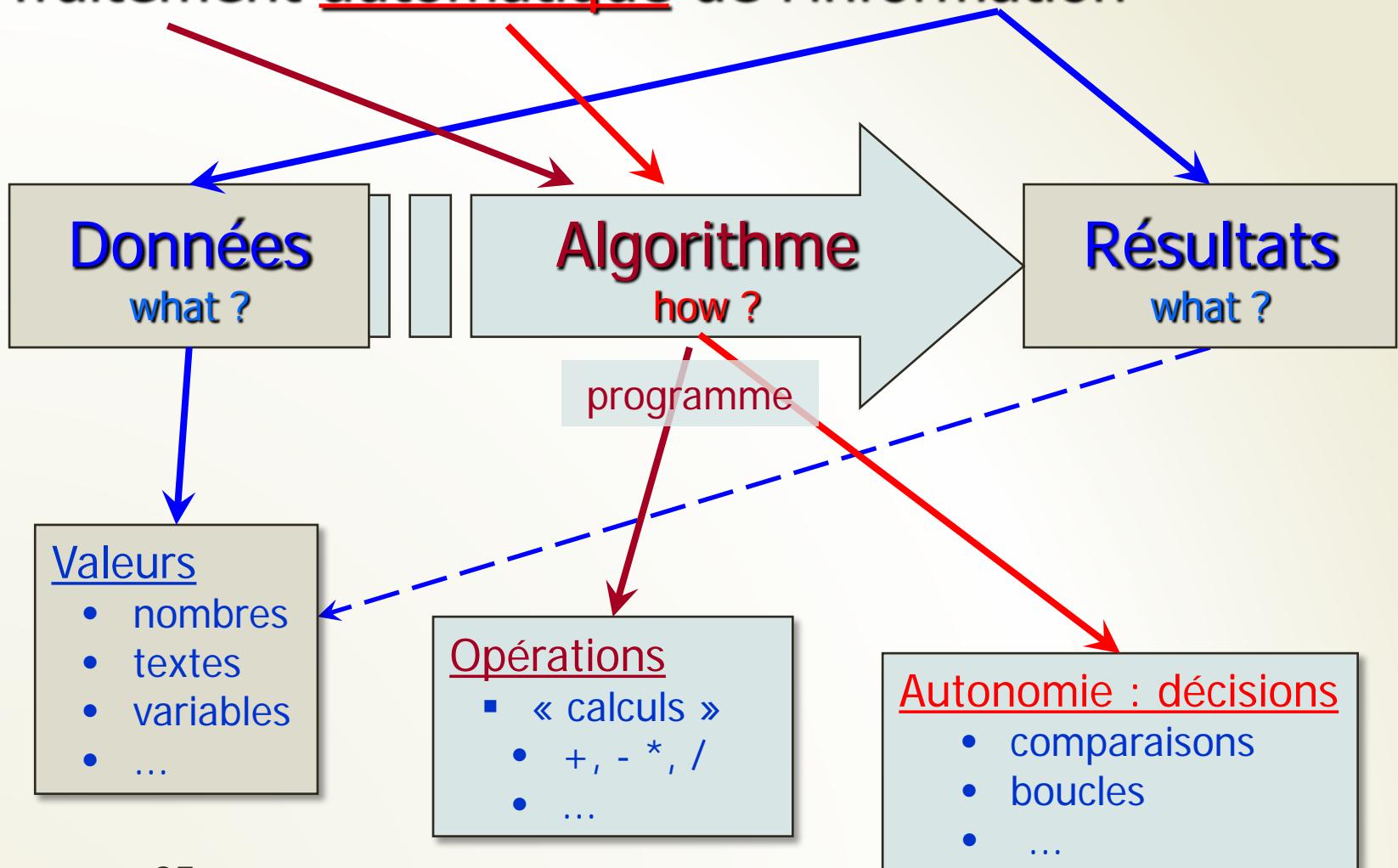
## LANGAGE → RÈGLES GRAMMATICALES ET SYNTAXIQUES (2B)

- il est recommandé de commenter le code pour augmenter sa lisibilité
  - les commentaires sont ignorés lors de l'exécution
  - deux formes de commentaires :
    - une ligne : commence par `//` jusqu'à la fin de la ligne  
ex. : `var a, b, c; // déclaration de variables`
    - plusieurs lignes : commence par `/*` et se termine par `*/`  
ex. : `/* script rédigé par ChL  
en septembre 2015  
*/`
- il est recommandé d'indenter le code pour faire ressortir sa structure (et augmenter sa lisibilité)  
*on y reviendra amplement*

## CONCEPTS FONDAMENTAUX :

TRAITEMENT  
AUTOMATIQUE  
DE L'INFORMATION

# Traitement automatique de l'information



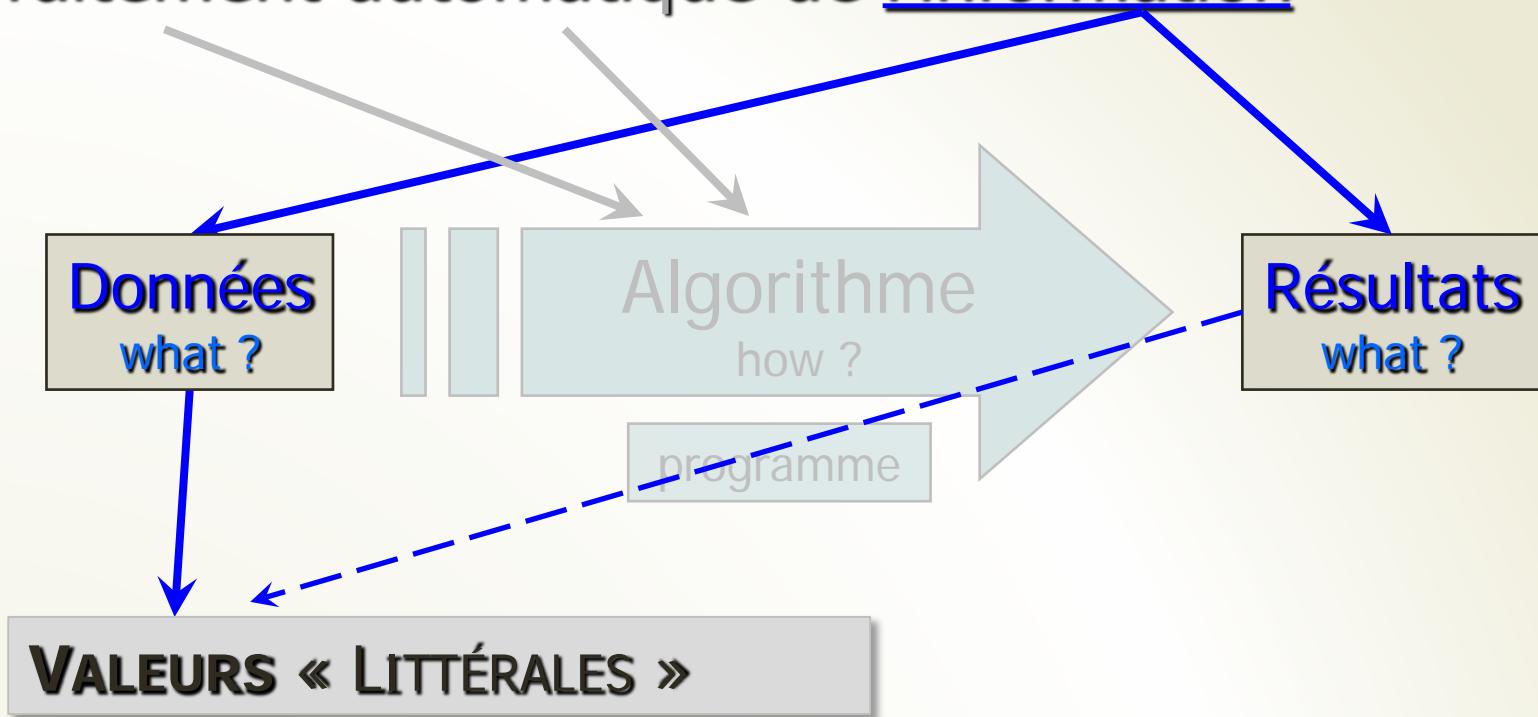
`x = 25;`

`y = 2.5;`

`z = x * y;`

`if (z > 100) { z = z * 0.9; }`

# Traitement automatique de l'information



## 1. VALEURS « LITTÉRALES » : COMMUNICATION « HUMAINE »

### □ NOMBRES : ÉCRITURE(S) USUELLE(S)

➤ ENTIERS :

23                    -2365                    0

➤ FRACTIONNAIRES (« RÉELS ») :

5.25                    -0.0365                    1.23E+03

### □ TEXTES : CARACTÈRES ET CHAÎNES "QUOTÉS"

➤ CARACTÈRES

'A'    'n'    ' '    ':'

➤ CHAÎNES DE CARACTÈRES

"Hello world !"                    "1348"

### 2. TYPE DE DONNÉES

- ENSEMBLE HOMOGÈNE DE VALEURS (UNIQUES)

1 est unique parmi les entiers

'A' est unique dans l'alphabet

"hello" est unique dans les chaînes de caractères

- ENSEMBLE D'OPÉRATEURS INTERNES

1 + 5 : le résultat est entier

"hello" + " world" : le résultat est un texte

- EXISTENCE D'UN ORDRE (→ COMPARAISONS, CONDITIONS ...)

-2 < -1 < 0 < 1 < 2 ...

'A' < 'B' < 'C' ...      'A' < 'a'      "papaye" < "pepsi"

- CODAGE ET REPRÉSENTATION PHYSIQUE (IMPLÉMENTATION)

## TYPES ÉLÉMENTAIRES SCALAIRES (EN ALGORITHMIQUE)

## ➤ TYPE ENTIER :

tous les entiers représentables

opérateurs : `+, -, *, quotient, reste, ()`

## ➤ TYPE RÉEL :

tous les fractionnaires représentables

opérateurs : `+, -, *, /, ()`

## ➤ TYPE CARACTÈRE :

tous les caractères du système

- utilisateur (`alphabets, chiffres, ponctuation, ...`)
- système (`esc, rtn, tab, ...`)

## ➤ TYPE TEXTE :

tous les textes représentables

opérateurs : `+` (concaténation)

et aussi le TYPE LOGIQUE  
cfr. plus loin ...  
... autonomie/décisions

## TYPES 'PRIMITIFS' (EN JAVASCRIPT)

## ➤ TYPE "NUMBER" :

**tous les nombres représentables** (domaine limité)

→ pas de distinction entre entiers et fractionnaires

**opérateurs** : +, -, \*, /, %, ()codage interne : 8 bytes (IEEE 754 standard)

## ➤ TYPE "STRING" :

**tous les textes représentables**

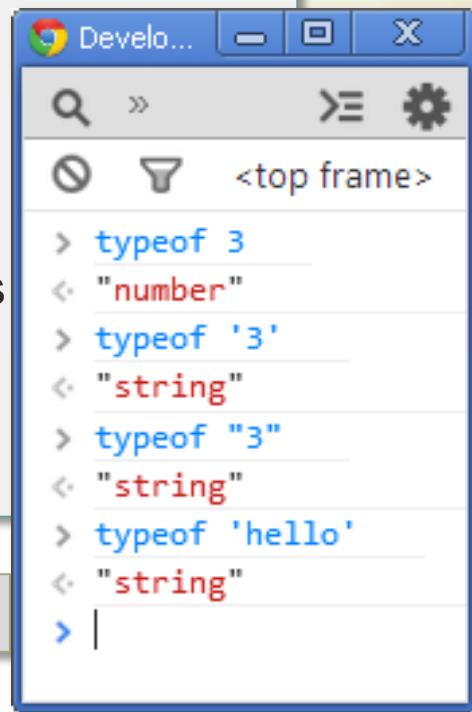
→ pas de distinction entre caractères et textes

**opérateur** : +codage interne : car. sur 2 bytes (UTF-8)

et aussi le TYPE "BOOLEAN"

cfr. plus loin ...

... autonomie/décisions

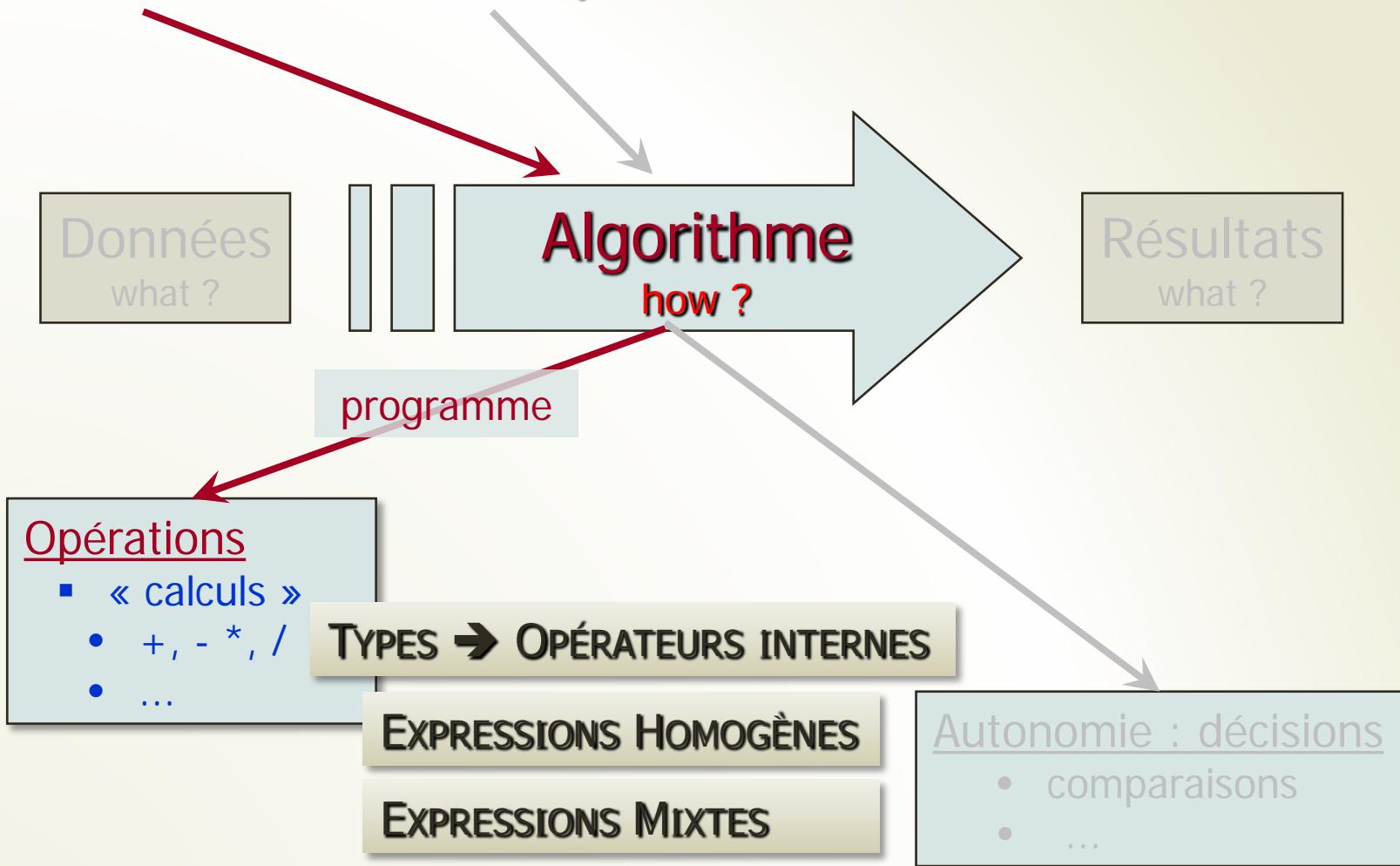
un opérateur utile : **typeof**


```

Devtools
<top frame>
> typeof 3
"number"
> typeof '3'
"string"
> typeof "3"
"string"
> typeof 'hello'
"string"

```

# Traitement automatique de l'information



appelées aussi  
opérandes

## 1. EXPRESSION HOMOGÈNE :

- ASSOCIATION DE VALEURS ET D'OPÉRATEURS

COMPATIBLES À L'INTÉRIEUR D'UN TYPE DONNÉ

`(1 + 5) * -3`

`"hello" + " " + "world"`

- TOUJOURS ÉVALUABLE : FOURNIT (RENOIE) UNE VALEUR DU TYPE CONSIDÉRÉ

`(1 + 5) * -3`

renvoie

`-18`



`111 / 0`

renvoie

`Infinity`

`"hello" + " " + "world"`

renvoie

`"hello world"`

valeur spéciale  
des "number"

I majuscule !!

## 2. EXPRESSION MIXTE :

- ASSOCIATION DE VALEURS ET D'OPÉRATEURS  
DE TYPES DIFFÉRENTS

**25 + "10"**      **???**  
**"333" - 111**    **???**  
**"11" / "2"**       **???**  
**"hello" \* 2**      **???**



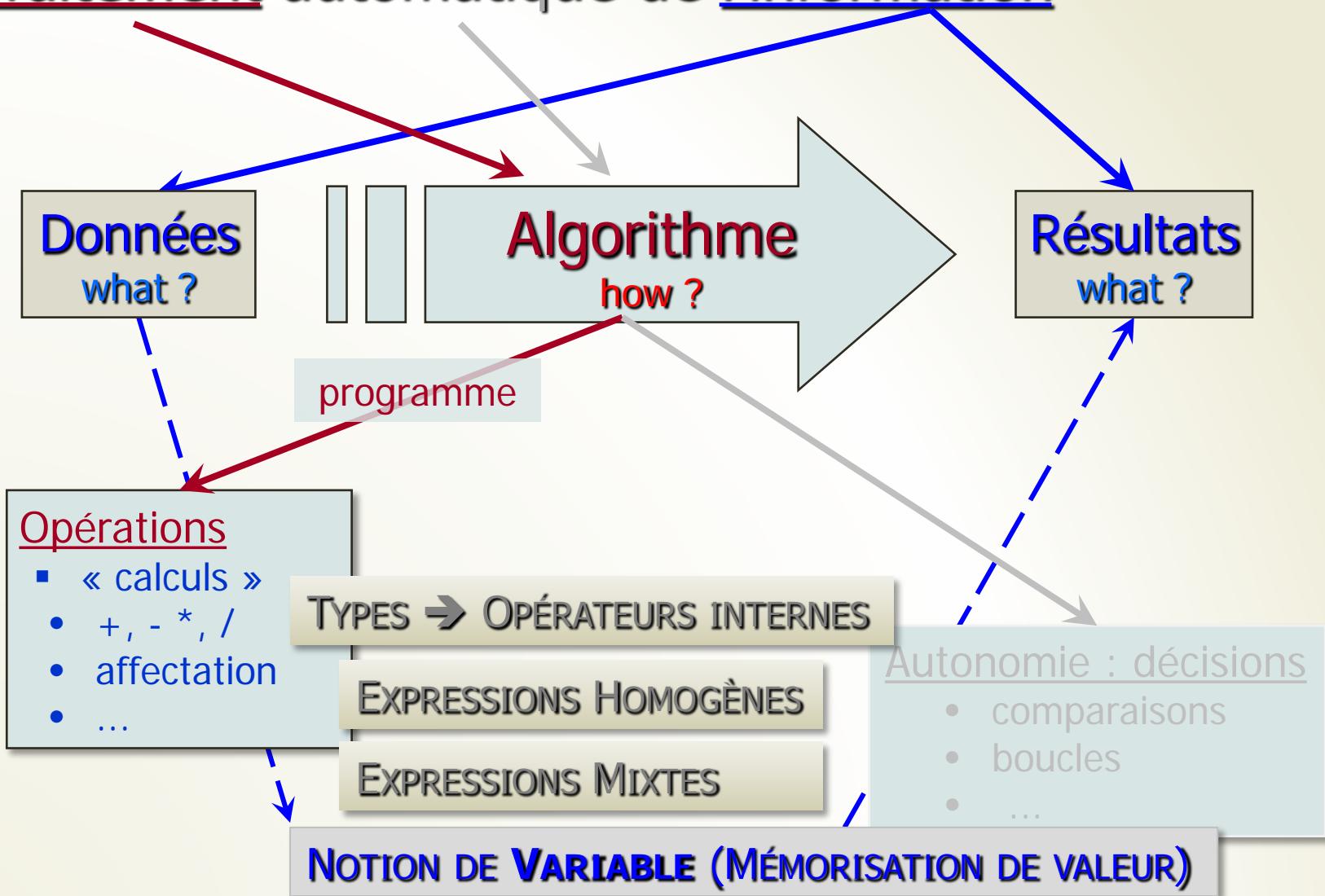
impossible en algorithmique  
et avec les langages 'classiques'

- TOUJOURS ÉVALUABLE : FOURNIT (RENVOIE) UNE VALEUR  
SELON DES RÈGLES PRÉCISES DE CONVERSION

<b>25 + "10"</b>	renvoie	<b>"2510"</b>	(number → string)
<b>"333" - 111</b>	renvoie	<b>222</b>	(string → number)
<b>"11" / "2"</b>	renvoie	<b>5.5</b>	(string → number)
<b>"hello" * 2</b>	renvoie	<b>NaN</b>	(not a number)



# Traitements automatique de l'information



## BUT : FOR-MA-LI-SER !

(UN PEU COMME EN MATHÉMATIQUES)

$10 * 12.5 * (1 + 0.06)$  expression explicite

$x * y * (1 + z)$  expression symbolique

### VARIABLE : NOM SYMBOLIQUE

- pour mémoriser une valeur (en mémoire centrale)
- et la réutiliser plus facilement dans des expressions

➤ DÉCLARATION : via le mot réservé **var**

```
var quantite, prixUnitaire, taxe;
```



👉 rappel : règles de nommage, écriture "camelCase"

## BUT : MÉMORISER UNE VALEUR DANS UNE VARIABLE

«nomDeVariable» = «valeur»;

```
var quantite, prixUnitaire, tva;  
var prixHTva, taxe, prixTvaC;  
  
quantite = 10;  
prixUnitaire = 12.5;  
tva = 0.06;  
  
prixHTva = quantite * prixUnitaire;  
taxe = prixHTva * (1 + tva);  
prixTvaC = prixHTva + taxe;
```

au sens large :

- valeur littérale
- variable
- expression
- retour de fonction

opérateur  
d'affectation

👉 rappel: notion d'instruction et ; final

plus tard : distinguer

- expression d'affectation
- instruction d'affectation

## EN JS, LES VARIABLES SONT "DYNAMIQUEMENT TYPÉES"

QU'EST-CE QUE CELA SIGNIFIE ?

- QUE C'EST L'AFFECTATION QUI DÉTERMINE LE TYPE
- QU'AU COURS DE "SA VIE" UNE MÊME VARIABLE PEUT RECEVOIR DES VALEURS DE TYPES DIFFÉRENTS
- LES VARIABLES SONT "SCALAIRES"
  - ELLES NE PEUVENT CONTENIR QU'UNE SEULE VALEUR
  - DONC L'AFFECTATION EST DESTRUCTRICE (EN TYPE/VALEUR)

```
var test;  
test = 10;           // typeof : "number"  
test = "hello";    // typeof : "string"  
test = 12.5;        // typeof : "number"  
test = 'A';         // typeof : "string"
```



## EN JS, LES VARIABLES SONT DYNAMIQUEMENT TYPÉES

DONC UNE QUESTION SE POSE :

QUEL EST LE TYPE D'UNE VARIABLE DÉCLARÉE MAIS NON ENCORE AFFECTÉE ?

➤ TYPE PRIMITIF "**undefined**" :

ne contient qu'une seule valeur : **undefined**

pas de " "  
ce n'est pas un string

toute expression numérique dont une opérande est **undefined** renvoie la valeur **NaN** (Not a Number)

```
var x;           // typeof x : "undefined"  
var y = 10;      // typeof y : "number"  
var z = x + y;  // typeof z : "number"  
                // z contient la valeur NaN  
y = undefined; // affectation légale
```



## !! RÔLE D'UNE VARIABLE SELON QUE SON NOM EST

- À GAUCHE DE L'OPÉRATEUR = (LVALUE)
- À DROITE DE L'OPÉRATEUR = (RVALUE)

l'expression à droite de = (la rvalue) est évaluée d'abord  
 et son résultat/valeur est affecté ensuite à la variable  
 spécifiée à gauche (la lvalue)

remplacer le contenu de test1  
 par son contenu actuel + 2

```
var test1;           // typeof : "undefined"
test1 = 10;          // typeof : "number"
test1 = test1 + 2;    // test1 = 10 + 2
                     // résultat : 12

var msg = "hello";    // déclaration/initialisation
msg = msg + " world"; // msg = "hello" + " world"
                     // résultat : "hello world"

var test2;           // typeof : "undefined"
test2 = test2 * 5.5;  // typeof : "undefined"
                     // résultat : NaN
```



## QUELQUES "SUCRES SYNTAXIQUES" D'AFFECTATION

**BUT : ÉCRITURE COMPACTE D'EXPRESSIONS D'AFFECTATION  
DANS LAQUELLE UNE VARIABLE EST À LA FOIS  
LVALUE ET RVALUE (À GAUCHE ET À DROITE DE L'OPÉRATEUR =)**

aussi bien sur des expressions homogènes que sur des expressions mixtes avec les mêmes conventions de conversion

expression	raccourci
<code>a = a + b</code>	<code>a += b</code>
<code>a = a - b</code>	<code>a -= b</code>
<code>a = a * b</code>	<code>a *= b</code>
<code>a = a / b</code>	<code>a /= b</code>
<code>a = a % b</code>	<code>a %= b</code>
<code>a = a + 1</code>	<code>a++ OU ++a</code>
<code>a = a - 1</code>	<code>a-- OU --a</code>

temporalité (on verra plus tard)

## PERMETTRE À L'UTILISATEUR DU SCRIPT D'AFFECTER À UNE VARIABLE UNE VALEUR SAISIE DEPUIS LE CLAVIER

**«nomDeVariable» = **prompt**(«invite»);**

**prompt()** et **alert()** sont des fonctions prédéfinies (du navigateur)

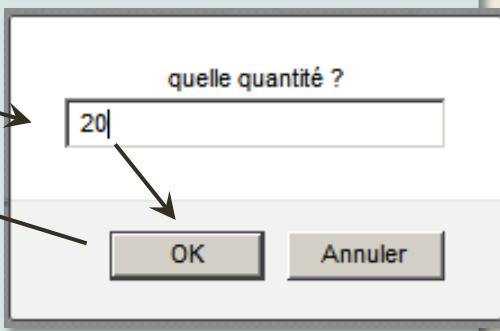
```
var quant, prixUnit, prixHtva, taxe, prixTvaC;
var tva = 0.06;
```



```
quant = prompt("quelle quantité ?");
prixUnit = prompt("quel prix unitaire ?");
```

```
prixHtva = quant * prixUnit;
taxe = prixHtva * tva;
prixTvaC = prixHtva + taxe;
```

```
alert('prix tvaC : ' + prixTvaC);
```

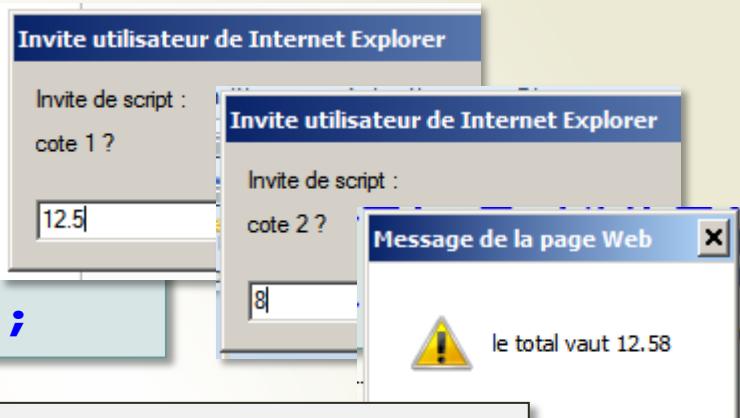


**prompt** renvoie toujours un "string"

## QUELQUES UTILITAIRES BIEN NÉCESSAIRES...

👉 prompt renvoie toujours un "string"

```
var cote1, cote2, somme;  
cote1 = prompt("cote 1 ?");  
cote2 = prompt("cote 2 ?");  
somme = cote1 + cote2;  
alert('le total vaut ' + somme);
```



- ➔ **parseInt(«string»)** : convertit un string en entier
- ➔ **parseFloat(«string»)** : convertit un string en décimal
- ➔ **+«string»** : convertit le "string" en "number", idem pour -

```
var quant;  
quant = parseInt('10 bouteilles'); // 10  
quant = parseInt('dix bouteilles'); // NaN  
quant = +'10.5'; // 10.5  
quant = +'10 bouteilles'; // NaN
```

## QUELQUES UTILITAIRES BIEN NÉCESSAIRES...

**ÉCRIRE DIRECTEMENT DANS LA PAGE AU LIEU D'UTILISER  
`alert()` QUI EST TROP LIMITÉ ET MODAL (BLOQUANT)**

**`document.write(«string»);`**

**EN ATTENDANT MIEUX (MANIPULATION DU DOM)**

**`document.write()` est en effet destructeur pour le contenu de la page ...**

**et on écrit du HTML (balises) !**

```
var cotel, cote2, somme;  
cotel = parseFloat(prompt("cote 1 ?"));  
cote2 = parseFloat(prompt("cote 2 ?"));  
somme = cotel + cote2;  
document.write('<p>Total : <b>' + somme + '</b></p>');
```

contenu de la variable  
converti en "string"



## ON MET LE TOUT DANS UNE PAGE (POUR VOIR) ...

```
<!DOCTYPE html>
<html>
  <head>
    <title>un script</title>
  </head>
  <body>
    <h1>un script dans une page</h1>
    <script>
      var cote1, cote2, somme, moyenne;
      cote1 = parseFloat(prompt("cote 1 ?"));
      cote2 = parseFloat(prompt("cote 2 ?"));
      somme = cote1 + cote2;
      moyenne = somme / 2;
      document.write('<p>Cote 1 : ' + cote1 + '</p>');
      document.write('<p>Cote 2 : ' + cote2 + '</p>');
      document.write('<p>-----</p>');
      document.write('<p>    Total : <b>' + somme + '</b></p>');
      document.write('<p> Moyenne : <b>' + moyenne + '</b></p>');
    </script>
  </body>
</html>
```

script01.html

## ON MET LE TOUT DANS UNE PAGE (POUR VOIR) ...



## QUELQUES UTILITAIRES BIEN NÉCESSAIRES...

**document.write()** peut

- être destructeur pour le contenu de la page ...
  - ne pas (bien) fonctionner avec certains navigateurs ...
- pour faire des tests rapides, il est plus simple d'envoyer les messages (texte brut non balisé) dans la console JavaScript du navigateur

**console.log(*«string»*)**

```
var cote1, cote2, somme;  
cote1 = parseFloat(prompt("cote 1 ?"));  
cote2 = parseFloat(prompt("cote 2 ?"));  
somme = cote1 + cote2;  
console.log('Total : ' + somme);
```

## ON ESSAIE EN FIREFOX (ARDOISE JAVASCRIPT) ...

The screenshot illustrates the use of the `console.log()` method instead of `document.write()` for outputting data to the browser's console.

**Top Window:**

- Code:

```
1 var cote1, cote2, somme, moyenne;
2 cote1 = parseFloat(prompt("cote 1 ?"));
3 cote2 = parseFloat(prompt("cote 2 ?"));
4 somme = cote1 + cote2;
5 moyenne = somme / 2;
6 document.write('<p>Cote 1 : ' + cote1 + '</p>');
7 document.write('<p>Cote 2 : ' + cote2 + '</p>');
8 document.write('<p>-----</p>');
9 document.write('<p> Total : <b>' + somme + '</b></p>');
10 document.write('<p> Moyenne : <b>' + moyenne + '</b></p>');
11 */
12 /*
13 Exception: The operation is insecure.
14 @6
15 */
```
- Output:

12.5

Exception: The operation is insecure.

**Bottom Window:**

- Code:

```
1 var cote1, cote2, somme, moyenne;
2 cote1 = parseFloat(prompt("cote 1 ?"));
3 cote2 = parseFloat(prompt("cote 2 ?"));
4 somme = cote1 + cote2;
5 moyenne = somme / 2;
6 console.log('<p>Cote 1 : ' + cote1 + '</p>');
7 console.log('<p>Cote 2 : ' + cote2 + '</p>');
8 console.log('<p>-----</p>');
9 console.log('<p> Total : <b>' + somme + '</b></p>');
10 console.log('<p> Moyenne : <b>' + moyenne + '</b></p>');
```
- Output:

<p>Cote 1 : 12.5</p>

<p>Cote 2 : 15</p>

<p>-----</p>

<p> Total : <b>27.5</b></p>

<p> Moyenne : <b>13.75</b></p>

**Console du navigateur:**

Scratchpad/2:6	Scratchpad/2:7	Scratchpad/2:8	Scratchpad/2:9	Scratchpad/2:10
<p>Cote 1 : 12.5</p>				
<p>Cote 2 : 15</p>				
<p>-----</p>				
<p> Total : <b>27.5</b></p>				
<p> Moyenne : <b>13.75</b></p>				

**Annotations:**

- A red box highlights the error message "Exception: The operation is insecure." in the top window, with a red arrow pointing to it from the text "pas de document.write()".
- A green box highlights the output of the `console.log()` statements in the bottom window, with a green arrow pointing to it from the text "mais bien console.log()".

## ON RÉSUME ?

### ➤ TYPES PRIMITIFS

- "number" (avec 2 valeurs spéciales : `Infinity` et `Nan`)
- "string"
- "undefined" 1 seule valeur : `undefined`

### ➤ OPÉRATEURS

- "number" `+, -, *, /, %, ()`
- "string" `+`
- affectation `=, +=, -=, *=, /=, %=, ++, --`  
`prompt()`
- type `typeof`

### ➤ CONVERSIONS "string" → "number"

- `parseInt()`
- `parseFloat()`
- `+, -`

### ➤ ÉCRITURE → UTILISATEUR

- `alert()`
- `document.write()`
- `console.log()`