

*Technologies
Web
(Javascript)*

DA/IA2
Henallux

Module 2

Variables et valeurs

> 2 + 2

< 4

> "2" + "2"

< "22"

> 2 + 2 - 2

< 2

> "2" + "2" - "2"

< 20



Au programme

➤ Les types de valeurs en Javascript

- number, string, boolean, function, object, undefined

➤ Déclarations de variables

➤ Les valeurs primitives

➤ Conversions entre les types de valeurs

- explicites ou implicites...

➤ Retour sur certaines opérations

Les types de valeurs

- Javascript est **non typé / faiblement typé / typé dynamiquement**.
 - Le contenu d'une variable peut changer de type au cours de l'exécution.
 - Mais, à chaque moment de son existence, une variable a un type précis.
- Exemple

```
let x = 7;           // x contient 7 (un nombre)
x += " nains";      // x contient "7 nains" (un string)
```
- **Implications**
 - On ne précise pas de type à la déclaration d'une variable.
 - De nombreuses conversions (implicites) sont effectuées.
 - Certaines erreurs de logique ne sont pas détectées ; il faut donc programmer de manière rigoureuse !

Les types de valeurs

- Obtenir le **type** d'une valeur / variable :
 - `typeof 12` donne **"number"**
 - `typeof true` donne **"boolean"**
 - `typeof "salut"` donne **"string"** } 3 types primitifs
- Autre type primitif : `undefined` (variable sans valeur)
 - `typeof undefined` donne **"undefined"**
 - Aussi en ES6 : les symboles (sort du cadre de ce cours).
- À côté de cela : des objets...
 - `typeof null` donne **"object"**
 - `typeof (new Date ())` donne **"object"**
 - `function double (x) { return 2*x; }` ou `const double = function (x) { return 2*x; }`
`typeof double` donne **"function"**

Déclarations de variables

Au programme de ce chapitre...

- La notion de scope
- Quatre types de déclarations de variables
- Le hoisting

Ensuite : *les valeurs primitives et les conversions*

La notion de scope (portée)

- Les variables utilisées dans un programme ont une certaine **durée de vie**, qui correspond à la partie du programme où celles-ci « existent » (c'est-à-dire peuvent être utilisées).

[langage C]

```
int triple (int x) {  
    int res = x * 3;  
    return res;  
}
```

Durée de vie
de **x** et de **res**

[Java]

```
for (int nb = 1 ; nb <= 10 ; nb++)  
    System.out.println(nb * val);
```

Durée de vie de **nb** ;
val a une durée de
vie plus large

- Le **scope** (ou **portée**) d'une variable = la portion de programme qui correspond à sa durée de vie.

La notion de scope (portée)

En C, chaque **scope** local correspond à un bloc délimité par { ... }.

```
#include <stdlib.h>
```

```
int nombre;
```

```
void table (int limite) {
```

facteur
variable locale

```
    int facteur = 1;
```

```
    while (facteur <= limite) {
```

res
variable locale

```
        int res = nombre * facteur;
```

```
        printf("%d x %d = %d", nombre, facteur, res);
```

```
        facteur++;
```

```
    }
```

```
}
```

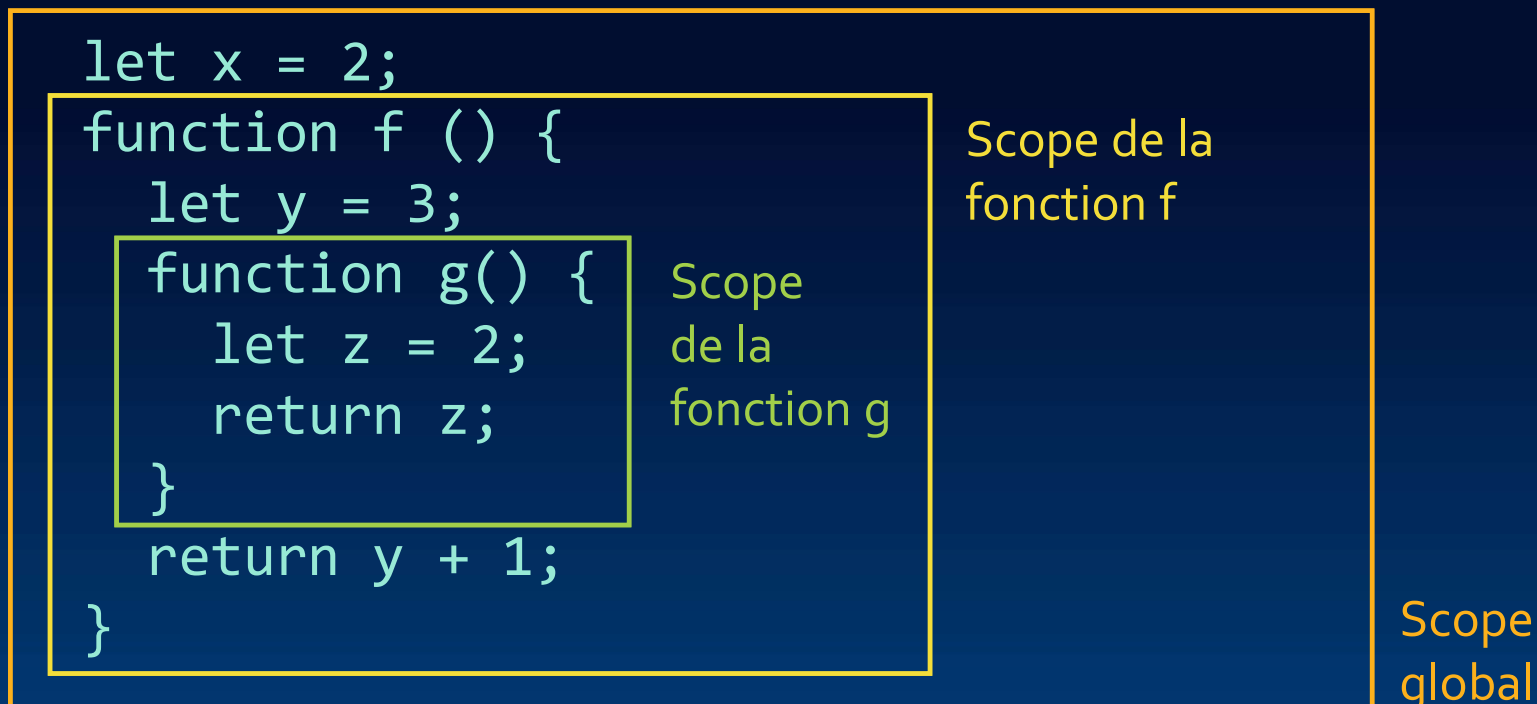
```
void main (void) {  
    scanf("%d", &nombre);  
    table(10);  
}
```

Dans un scope, on connaît toutes les variables locales ainsi que les variables des scopes englobants.

nombre
variable
globale

La notion de scope (portée)

- Javascript possède également un scope global et des scopes locaux. Les scopes sont emboîtés (structure d'arborescence).



La notion de scope (portée)

- Dans un scope, on connaît
 - les variables/fonctions locales et
 - les variables/fonctions des scopes englobants.

```
function f() {  
  let chance = 7;  
  function g() {  
    alert(chance);  
  }  
  g();  
}
```

La variable chance est déclarée dans ce scope.

On peut accéder à chance ici.

```
f();  
alert(chance);
```

Mais pas ici :
ReferenceError

Déclaration de variables

Javascript autorise **4 types de déclarations de variables**.

- (1) **Pas de déclaration** : variable directement utilisée

```
msg = "hello";
```

- (2) **Déclaration avec let** (ES6)

```
let msg = "hello";
```

- (3) **Déclaration avec const** (ES6)

```
const msg = "hello";
```

- (4) **Déclaration avec var**

```
var msg = "hello";
```

Déclaration de variables

- (1) **Pas de déclaration** : la variable est utilisée/initialisée sans déclaration.

→ Variable déclarée au niveau global

```
function ditBonjour () {  
    msg = "hello";           // globale !  
    alert(msg);  
}  
ditBonjour();  
alert(msg);                 // la variable existe ici !
```

- Si la variable est utilisée en lecture sans être initialisée, erreur à l'exécution (**ReferenceError**) !

```
console.log(msgPasInitialisé); // ReferenceError
```

- Si la variable existe dans un scope englobant, on ne la redéclare pas !

Déclaration de variables

- (2) [ES6] **Déclaration avec let**
→ Variable locale au scope du bloc (ou globale si on ne se trouve pas dans un bloc)

```
function ditBonjour () {  
  let iter = 1;           // locale à ditBonjour  
  while (iter < 5) {  
    let msg = "hello" + iter // locale au bloc  
    console.log(msg);  
    iter++;  
  }  
}  
let nombre = 7;           // globale (car hors-bloc)
```

- Version la plus proche de la déclaration de variables en C/Java.
- On ne peut pas redéclarer avec let une variable déjà déclarée.

Déclaration de variables

- (3) [ES6] **Déclaration avec const**

→ Constante locale au scope du bloc (ou globale si on ne se trouve pas dans un bloc), **doit être initialisée**

```
function ditBonjour () {  
  let iter = 1;      // erreur si const car modifiée ensuite  
  while (iter < 5) {  
    const msg = "hello" + iter; // locale au bloc  
    console.log(msg);  
    iter++;  
  }  
}  
const nombre = 7;    // globale (car hors-bloc)
```

- Il faut donner une valeur (initiale et finale) dans la déclaration !
- On ne peut pas redéclarer une variable déjà déclarée.

Déclaration de variables

Syntaxe « historique »
pour la rétrocompatibilité
(sans doute à éviter)

- (4) **Déclaration avec var**

→ Variable locale au scope de la fonction (ou globale si on ne se trouve pas dans une fonction)

```
function ditBonjour () {  
    var msg;           // locale à ditBonjour  
    msg = "hello";  
    alert(msg);  
}
```

```
var nombre = 7;        // globale (car hors-fonction)
```

- Attention : le scope est la fonction englobante, pas le bloc !
- On peut « redéclarer » une variable déjà déclarée (la 2^e déclaration est ignorée [mais pas l'initialisation]).

Déclaration de variables

- *Que vont produire les appels $f(7)$ et $f(-3)$? $g(7)$ et $g(-3)$?*

```
function f(x) {  
  var tmp = 4;  
  if (x > 0) {  
    var tmp = 5;  
    alert(tmp);  
  }  
  alert(tmp);  
}
```

```
function g(x) {  
  let tmp = 4;  
  if (x > 0) {  
    let tmp = 5;  
    alert(tmp);  
  }  
  alert(tmp);  
}
```

Déclaration de variables

- *Que vont produire les appels $f(2)$ et $g(2)$?*

```
function f(x) {  
  for (var i = 0 ; i <= x ; i++)  
    console.log(i);  
  console.log(i);  
}
```

```
function g(x) {  
  for (let i = 0 ; i <= x ; i++)  
    console.log(i);  
  console.log(i);  
}
```

Exercice

- Exemple : que produit le code suivant ?

```
alert(x);           // ReferenceError : aucun x connu
function f () {
  x = 17;           // déclare f mais ne l'exécute pas !
}
alert(x);           // ReferenceError : aucun x connu
f();                // f déclare x au niveau global
alert(x);           // x est connu au niveau global (et vaut 17)
```

Et si on avait utilisé let ou var à la 3^e ligne de code ?

ReferenceError la 3^e fois aussi car x n'est pas global

Exercice

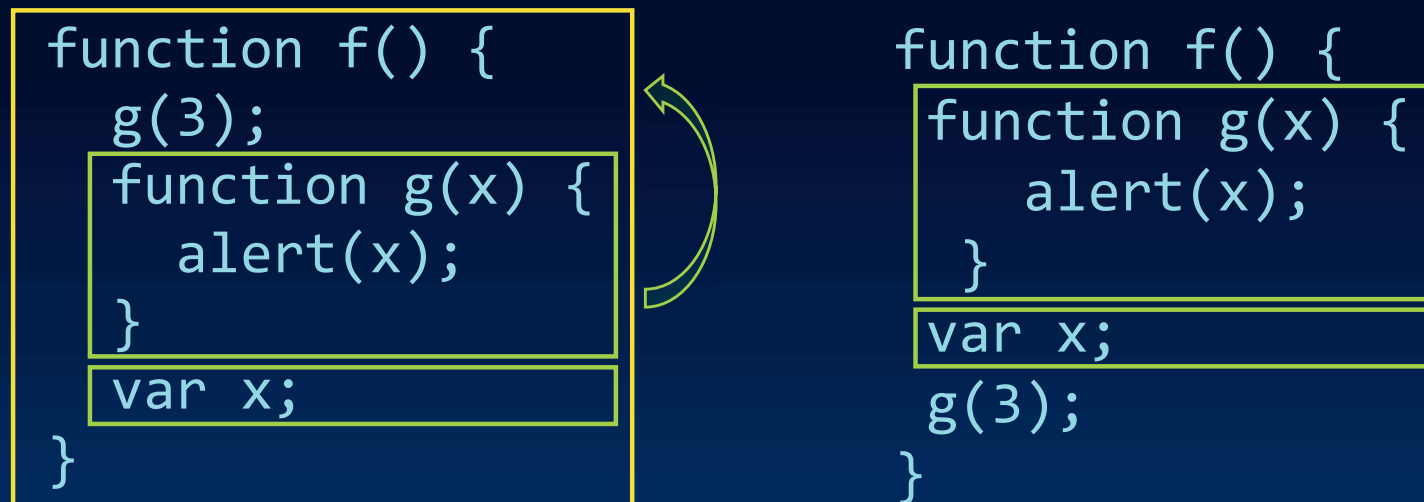
- Exemple : que produit le code suivant ?

```
function troisFois () {  
    for (i = 0 ; i < 3 ; i++)  
        console.log("Bonjour !");  
}  
function nFois (n) {  
    for (i = 0 ; i < n ; i++)  
        troisFois();  
}  
nFois(2);
```

- Et avec l'appel `nFois(4)` ?
- Et avec `nFois(5)` ?

Hoisting (hissage)

- Dans chaque scope fonctionnel, les **déclarations de variables via var** (et les déclarations de fonctions) sont **hissées** vers le début.
- Exemple :



- Concrètement, cela signifie qu'on peut utiliser une fonction même si sa définition se trouve plus loin dans le code.

Hoisting (hissage)

- Exemples (déclarations de fonctions) :

```
function go() {  
  alert("ok");  
}  
go();
```

ok

```
go();  
function go() {  
  alert("ok");  
}
```

ok

Ces deux bouts de programme
sont équivalents.

```
go();  
function go() {  
  alert("ok");  
}  
function go() {  
  alert("ko");  
}
```

ko

Hoisting (hissage)

- Exemples (déclarations de variables) :

```
function f () {  
  var x = 3;  
  alert(x);  
}  
f();
```

3

```
function f () {  
  alert(x);  
}  
f();
```

(ReferenceError)

```
function f () {  
  alert(x);  
  var x = 3;  
}  
f();
```

undefined

```
function f () {  
  var x;  
  alert(x);  
  x = 3;  
}  
f();
```



Seule la déclaration est hissée !
Pas l'initialisation

Hoisting (hissage)

- Et dans le cas de « **let** » et « **const** » ?
 - Hissage vers le début du bloc
 - Mais on ne peut pas utiliser la variable avant sa déclaration !
(C'est ce qu'on appelle la TDZ ou Temporal Dead Zone)
- Exemple : que produit un appel f() ?

```
var x = 7;  
function f() {  
  console.log(x);  
  let x = 3;  
  console.log(x);  
}
```

```
f()  
ReferenceError: can't access  
lexical declaration `x' before  
initialization
```

```
var x = 7;  
function f() {  
  console.log(x);  
  var x = 3;  
  console.log(x);  
}
```

```
f()  
undefined  
3
```

Autres exercices dans le labo

- Dans un premier temps, lisez le code et essayer de déterminer le résultat qui sera affiché.
- Ensuite seulement, **testez** dans la console Firefox.
- Observez les résultats et assurez-vous de **comprendre** pourquoi Javascript produit ces résultats !

Comparaison des déclarations

	let	const	var	(sans)	fonction
Scope	bloc	bloc	fonction	global	fonction*
Hoisting	TDZ	TDZ	déclaration	non	oui
Utilisation avant décl	erreur	erreur	undefined	erreur	—
Redéclaration	erreur	erreur	accepté	(impossible)	accepté

(*) *scope fonctionnel par défaut, bloc en mode strict*

- **En pratique** [Clean Code],
 - éviter **var** et les déclarations implicites (complexes, peu lisibles) ;
 - préférer **const** (+ initialisation !) si la valeur ne change pas ;
 - sinon, utiliser **let**.

Les valeurs primitives

Au programme de ce chapitre...

➤ **Les nombres**

➤ **Les booléens**

➤ **Les chaînes de caractères**

- *Comment écrire les valeurs de ce type (littéraux) ?*
- *Quelles sont les opérations utilisables ?*

Ensuite : *Conversions explicites et implicites*

Littéraux pour les nombres

- **Valeurs standards**

- **Entiers** : 42 -78
- **Réels** : 342.17 1.36E78
- Autres bases : 037 (octal) 0x3BFF (hexa)
[ES6] 0o767 (octal) 0b11101010 (binaire)
- tous codés sur 64 bits (sans distinction entier/réel)

- **Valeurs spéciales**

- Infinity : résultat de 5/0
- -Infinity
- NaN (Not A Number) : résultat de 100/"Hello"
- Pour les repérer :
 - isFinite(x) \equiv x a une valeur numérique standard
 - isNaN(x) \equiv x vaut NaN
 - Note : si x n'est pas un nombre, il est tout d'abord converti en nombre.

Opérations sur les nombres

- **Opérateurs usuels**

- Opérations binaires : `+` `-` `*` `/` `%` `**` (exposant)
- Opérations unaires : `-` `++` `--`
- Opérations de comparaison : `<` `<=` `>` `>=` `==` `===`
- *Modulo sur les réels* : `4.7 % 1.3` donne `0.8`

- **Opérateurs bit à bit** : `&` `|` `~` `^` `<<` `>>` `>>>`

- Outils de la « **bibliothèque** » **Math**

- Constantes : `Math.PI`, `Math.E`, `Math.SQRT2`...
- `Math.abs(x)`, `Math.pow(x,y)`, `Math.sin(x)`, ...
- `Math.floor(x)`, `Math.ceil(x)`, `Math.round(x)`
- `Math.max(x,y,...)`, `Math.min(x,y,...)`
- `Math.random()` donne un nombre entre 0 (compris) et 1 (exclu)

[v11, juin 2020] Les BigInt

- **Nouveau type primitif : BigInt**

- Objectif : représenter des entiers sans limite de taille (on peut aller au-delà de l'entier maximal `Number.MAX_SAFE_INTEGER`).
- Littéraux : `9007199254740992n`
- Opérations : `+` `-` `*` `**` `/` `%` `<` `<=` `>` `>=` `==` `===`
mais pas compatible avec les méthodes de Math (ex : `Math.pow`)
- Différents des nombres, mais comparables avec eux :
 - `2n == 2` → `true`
 - `2n === 2` → `false`
 - `2n < 5` → `true`
- Fonctionnent généralement comme les nombres pour le casting implicite

Les booléens

- **Littéraux** pour les booléens : `true` `false`
- **Opérations** sur les booléens
 - Opérations standards : `!` `&&` `||`
avec une signification particulière (voir plus tard)
 - Opérateur ternaire : `? :`
`nbElem + " élément" + (nbElem > 1 ? "s" : "")`

Littéraux pour les strings

- **Chaînes standards**

- Encadrées par des guillemets ou des apostrophes (au choix)
 - "pommes", 'chaîne', "aujourd'hui", 'aujourd\'hui'
 - Utile : `document.write('<p class="intro">...</p>')`;
- Caractères échappés : `\n` `\t` `\'` `\"` `\\`
- Caractères selon leur code : `\xA5` ou `\x12B6` (hexa), `\123` (octal)

Littéraux pour les strings

- [ES6] Gabarits ou **template literals**
 - Encadrées par des apostrophes inverses (alt+96)
 - Caractère échappé (en plus des standards) : `\``
 - Peuvent être réparties sur plusieurs lignes
 - ``le début d'un texte qui
continue plus bas``
 - Peuvent inclure des expressions à évaluer
 - ``la somme vaut ${nb1+nb2}, cher ${nom}.``
 - `document.write(``
- [ES6] Pour aller plus loin : *tagged template literals*
 - Permettent de modifier composante par composante un template literal.

Opérations sur les strings

- **Opérations standards**

- **Concaténation** : "Bonjour, " + nom
- **Longueur** : nom.length
- **Extraction** d'un caractère : nom[0] ou s.charAt(0)

- Les chaînes de caractères sont **immuables** (immutable).

```
let s = "bingo";  
s.length;      // donne 5  
s[1];          // donne "i"  
s.length = 4;  // sans effet  
s[1] = "a";    // sans effet  
s;             // vaut toujours "bingo"
```


Opérations sur les strings

- Nombreuses autres **opérations prédéfinies**

- **Tests :**

```
s.startsWith(deb)  
s.endsWith(fin)  
s.includes(partie)
```

- **Extraction :**

```
s.substr(deb, longueur)  
s.substring(deb, fin)
```

- **Recherche :**

```
s.indexOf(partie)  
s.lastIndexOf(partie)
```

- **Décomposition :**

```
s.split(sep)
```

Conversions entre types

Au programme de ce chapitre...

➤ Conversions explicites

➤ Règles de conversion

- *vers un nombre*
- *vers un string*
- *vers un booléen*

➤ Conversions implicites

- *pour l'opérateur +*
- *pour les opérateurs de comparaison == et ===*
- *pour les autres opérateurs de comparaison (<, <=, >, >=)*

Ensuite : *Autres opérations*

Conversions explicites

- Pour **convertir vers un type primitif**, on peut utiliser les trois fonctions suivantes :

```
Number("127") // donne 127  
String(true)  // donne "true"  
Boolean(0)    // donne false
```

- Note : ces fonctions s'écrivent avec une majuscule.
- Pourquoi ? Parce que ce sont des constructeurs (voir plus tard, dans le module sur les objets).
- Mais comment la conversion s'effectue-t-elle ?
Number("15+12")
Boolean(-1)
String(undefined)
→ **Règles de conversion** (3 transparents suivants)

Règles de conversion (1/3)

- **Vers un nombre** (`Number(x)` ou `+x` par exemple) :

Valeur	Conversion en nombre	Exemple
nombre	inchangé	<code>Number(3) → 3</code>
booléen	<code>true → 1</code> <code>false → 0</code>	<code>Number(true) → 1</code> <code>Number(false) → 0</code>
chaîne	<code>""</code> (vide ou blancs) → 0 <code>"nombre"</code> → nombre (vide/blancs ignorés) autres chaînes → NaN	<code>Number("\t \n") → 0</code> <code>Number(" 42\t") → 42</code> <code>Number("7nains") → NaN</code>
undefined	NaN	<code>Number(undefined) → NaN</code>
objet	<code>null → 0</code> autres objets → via <code>valueOf()</code>	

Règles de conversion (2/3)

- **Vers un booléen** (`Boolean(x)`, `!!x`, `x?:` ou `if(x)` par exemple)

Valeur	Conversion en booléen	Exemple
Nombre	0 et NaN → false autres nombres → true	<code>Boolean(0) → false</code> <code>Boolean(127) → true</code>
booléen	inchangé	<code>Boolean(true) → true</code>
chaîne	"" (vide) → false autres chaînes → true	<code>Boolean("") → false</code> <code>Boolean("vrai") → true</code>
undefined	false	<code>Boolean(undefined) → false</code>
objet	null → false autres objets → true	

- Les valeurs correspondant à false sont dites « **falsy** ».
 - `false`, `undefined`, `null`, `0`, `NaN` et `""`
- Les autres sont dites « **truthy** ».

Règles de conversion (3/3)

- **Vers un string** (`String(x)` ou `" " + x` par exemple)

Valeur	Conversion en string	Exemple
Nombre	NaN → "NaN" Infinity → "Infinity" autres → leur écriture	<code>String(17.4)</code> → "17.4" <code>String(1E3)</code> → "1000"
booléen	true → "true" false → "false"	<code>String(true)</code> → "true" <code>String(false)</code> → "false"
chaîne	inchangé	<code>String(" 3 ")</code> → " 3 "
undefined	"undefined"	
objet	null → "null" autres objets : via <code>toString()</code>	

Conversions explicites

- Deux autres méthodes pour convertir en nombres :

- **parseFloat(s)** convertit s en string, supprime les blancs initiaux et transforme en nombre le plus long préfixe possible

<code>parseFloat(true)</code>	→ <code>parseFloat("true")</code> → NaN
<code>parseFloat("")</code>	→ NaN
<code>parseFloat("123piano")</code>	→ 123
<code>parseFloat(" \t45-4")</code>	→ 45
<code>parseFloat(" 17.12")</code>	→ 17.12

- **parseInt(s,base)** : idem mais pour un entier dans la base donnée

<code>parseInt(true, 6)</code>	→ NaN
<code>parseInt(" 17.12")</code>	→ 17
<code>parseInt(" 17.12", 8)</code>	→ 15 (= 17 ₈)

- Dans les deux cas, on s'arrête au premier caractère illégal par rapport à ce qu'on recherche.

Conversions implicites

- Certaines opérations n'ont de sens que pour un type de valeurs.
 - L'opération $x*y$: si, à l'exécution, x et y contiennent autre chose que des nombres, Javascript les convertit implicitement en nombres.
 - `true * "-3" → 1 * (-3) → -3`
 - `false * "true" → 0 * NaN → NaN`
 - Idem dans le cas d'une condition : conversion vers un booléen
 - `if ("ok") ... → if (true) ...`
- Mais que se passe-t-il dans les cas plus ambigus comme `"27" + 2` ?
- D'où l'utilité de connaître l'existence des **règles de calcul** pour chaque opération (*mais pas forcément de les connaître par cœur*) !
 - (1) Comment Javascript convertit-il ?
 - (2) Quand Javascript convertit-il ?

Conversions pour +

- Quand JS convertit-il pour **l'opérateur +** ?
 - **au moins 1 string** : convertir en strings et concaténer
 - **sinon** : convertir en nombres et additionner
- Exemples
 - $3 + "2" \rightarrow "32"$
 - $3 - "2" \rightarrow 1$
 - $"15" + "1" \rightarrow "151"$
 - $7 + \text{true} \rightarrow 8$

Conversions pour ==

- Quand JS convertit-il pour **l'opérateur ==** ?
 - **valeurs de même type** : comparaison simple
 - Mais NaN n'est égal à personne, pas même à lui-même !
 - **sinon**, s'il y a au moins un `null` ou un `undefined`,
 - `null == undefined` → `true`
 - tous les autres → `false`
 - **sinon**, tout convertir en nombres
- Exemples
 - `"4" == true` → `false` `"" == "0"` → `false`
 - `5 == "5"` → `true` `0 == "0"` → `true`
 - `'2' == 2` → `true` `false == undefined` → `false`
 - `"\t\n" == 0` → `true` `true == null` → `false`
 - `0 == ""` → `true` `false == null` → `false`

Conversions pour ===

- Quand JS convertit-il pour l'opérateur === ?
 - Jamais !
 - donne automatiquement false si les types sont différents
- Exemples
 - `33 === "33" → false`
 - `null === undefined → false`
 - `NaN === NaN → false`

Conseil : préférez === à == pour éviter les mauvaises surprises dues aux conversions automatiques !

Conversions pour <

- Quand JS convertit-il pour **l'opérateur <** ?
 - **si 2 strings** : comparaison lexicographique
 - **sinon** : convertir en nombres et comparer
- Exemples
 - `31 < "274" → true`
 - `"147" < "25" → true`
 - `"150" < 99 → false`

Conversions implicites

- Les conversions implicites rendent le **code difficile à comprendre**.
 - règles complexes
 - règles difficiles à retenir quand on jongle avec plusieurs langages
- **[Clean Code]** Donc, il vaut mieux **éviter** de les utiliser !
 - *Pourquoi les apprendre ?*
Parce qu'il faut aussi être capable de lire du « mauvais » code.
Parce que cela peut aider à trouver la source de certaines erreurs.
 - *Comment les éviter ?*
En convertissant explicitement (Number, String, Boolean).
- **[Clean Code]** Aussi, éviter ==
 - Utiliser plutôt === pour éviter les mauvaises surprises.
 - Exemple : au lieu de `val == 123`, utiliser `Number(val) === 123`

Retour sur les opérations

Au programme de ce chapitre...

- **Affectation**
- **Opérateurs typeof et void**
- **Opérateurs logiques || et &&**
 - *Définition étendue à tous les types*

Affectations

- **Version standard**

```
vitesseKmH = 90;
```

- **Versions raccourcies** (aussi `*=`, `/=`, `-=` etc.)

```
nbElements++;
```

```
multiple += 5;
```

- C'est une expression qui renvoie la valeur affectée

```
[Moche] valeur = 5;  
         console.log(valeur = 7);
```

```
[Acceptable] x = y = z = 7;
```

Opérateurs typeof et void

- L'opérateur **typeof**
 - Indique le type de son argument
 - Si l'argument référence une variable qui n'existe pas, renvoie **'undefined'**.
 - ```
if (typeof x === 'undefined') ...
 // si x est undefined ou n'existe pas
```
- L'opérateur **void**
  - Il évalue son argument mais ne renvoie pas sa valeur.
  - Deux exemples d'utilisation
    - ```
<a href="javascript:void(0)">Ne rien faire</a>
```
 - ```

 Envoyer le formulaire
```

# Retour sur l'opérateur `||`

- Sémantique de l'opération logique **`A || B`**
  - On évalue l'expression A en a.
  - **Si a converti en booléen donne true**, le résultat est a.
  - **Sinon**, le résultat est la valeur de B.
- Exemples

```
(12 + 5) || "erreur"
→ 17 || "erreur"
→ 17
```

```
"" || 33
→ 33
```

(Rappel) valeurs falsy : `false`, `undefined`, `null`, `0`, `NaN` et `""`

# Retour sur l'opérateur ||

- Utilité (mais il y a mieux) : **définir des valeurs par défaut.**

- Exemple :

```
function aboyer (nomChien) {
 nomChien = nomChien || "Fido";
 alert(nomChien + " aboie !");
}
```

- Si `nomChien` contient une valeur (par exemple une chaîne de caractères non vide), la variable reste inchangée.
- Mais son contenu devient "Fido" si `nomChien` est `0`, une chaîne vide, `null` ou `undefined` (valeurs « falsy »).
- Impossible donc d'utiliser cette "astuce" dans les cas où une de ces valeurs (`0`, chaîne vide, `null` ou `undefined`) serait admissible !  

```
nbDoigts = nbDoigts || 10;
```

# Retour sur l'opérateur &&

- Sémantique de l'opération logique **A && B**
  - On évalue l'expression A en a.
  - **Si a converti en booléen donne false**, le résultat est a.
  - **Sinon**, le résultat est la valeur de B.
- Exemples

```
(12 + 5) && "ok"
→ 17 && "ok"
→ "ok"
```

```
undefined && 33
→ undefined
```

# Retour sur l'opérateur &&

- Utilité : écrire des conditions d'existence.
- Exemple :
  - `if (popup && popup.visible) ...`
  - Si aucun objet `popup` n'existe, la condition s'évalue à `undefined` et donc à `false` ; le code n'est pas exécuté.
  - Par contre, si `popup` existe, on teste alors le booléen `popup.visible` et, s'il est vrai, on exécute le code.
    - Risque de plantage si `if (popup.visible) ...`
- Autre exemple :
  - `init && init();`
  - Si aucun objet (aucune fonction) `init` n'existe, l'expression renvoie `undefined` (qui est ignoré).
  - Par contre, si `init` existe, cette fonction est alors exécutée.