


Le langage Javascript (1^{re} partie : les bases)

HENALLUX

Technologies web

IG2 – 2015-2016

Au programme...

- 
- **La syntaxe de Javascript**
 - Règles générales
 - **Les valeurs manipulables en Javascript**
 - Types de valeurs
 - Littéraux (comment écrire ces valeurs)
 - Opérations
 - Conversions explicites et implicites
 - **Les instructions en Javascript**
 - **Les fonctions en Javascript**
 - Définition de fonctions
 - Fonctions anonymes
 - Variables locales et variables globales
 - Hoisting

La syntaxe de Javascript

- Javascript ignore les blancs.
 - Blanc = espace, tabulation, retour à la ligne...
 - comme HTML
 - autorise une indentation claire [clean code]
- À l'inverse du clean code : les scripts minimalisés
 - Noms de variables aussi courts que possibles
 - aucun blanc inutile
 - illisible par un humain
 - mais permet d'obtenir un fichier plus petit

La syntaxe de Javascript

- Un exemple (extrait) de code minimaliste
 - À éviter dans le cadre de ce cours !

```
ig"),db=function(a,b,c){var d="0x"+b-65536;return
d!=d||c?b:0>d?String.fromCharCode(d+65536):String.fromCharCode(d>>10|5529
6,1023&d|56320)};try{I.apply(F=J.call(v.childNodes),v.childNodes),F[v.chil
dNodes.length].nodeType}catch(eb){I={apply:F.length?function(a,b){H.apply(
a,J.call(b))}:function(a,b){var
c=a.length,d=0;while(a[c++]=b[d++]);a.length=c-1}}function
fb(a,b,d,e){var
f,h,j,k,l,o,r,s,w,x;if((b?b.ownerDocument||b:v)!=n&&m(b),b=b||n,d=d||[],!
a||"string"!=typeof a)return
d;if(1!==(k=b.nodeType)&&9!==(k))return[];if(p&&!e){if(f=_.exec(a))if(j=f[1]
){if(9===k){if(h=b.getElementById(j),!h||!h.parentNode)return
d;if(h.id===j)return d.push(h),d}else
if(b.ownerDocument&&(h=b.ownerDocument.getElementById(j))&&t(b,h)&&h.id===
j)return d.push(h),d}else{if(f[2])return
I.apply(d,b.getElementsByTagName(a)),d;if((j=f[3])&&c.getElementsByName
e&&b.getElementsByClassName)return
I.apply(d,b.getElementsByClassName(j)),d}if(c.qsa&&(q||!q.test(a))){if(s=
r=u,w=b,x=9===k&&a,1===k&&"object"!=b.nodeName.toLowerCase()){o=g(a),(r=b
.getAttribute("id"))?s=r.replace(bb,"\\$&"):b.setAttribute("id",s),s="[id=
'+s+']" ,l=o.length;while(l--
)o[l]=s+qb(o[l]);w=ab.test(a)&&ob(b.parentNode)||b,x=o.join(",")}if(x)try{
return
```

La syntaxe de Javascript

- Écrire des **commentaires** en Javascript
 - Syntaxe similaire au C et à Java
 - **//** pour le reste de la ligne
 - **/* ... */** pour plusieurs lignes / une partie de ligne
- À propos des **identificateurs** en Javascript :
 - commencent par une lettre, **\$** ou **_**
 - composés de lettres, chiffres, **\$** et **_**
 - (sauf les mots réservés)
 - (éviter les **\$** et **_** en début : variables prédéfinies)
- Javascript est **sensible à la casse** !
 - **mavariabLe** ≠ **maVariable** ≠ **MAVARIABLE**

La syntaxe de Javascript

- Liste des mots réservés en Javascript

abstract	debugger	final	instanceof	public	transient
boolean	default	finally	int	return	true
break	delete	float	interface	short	try
byte	do	for	long	static	typeof
case	double	function	native	super	var
catch	else	goto	new	switch	void
char	enum	if	null	synchronized	volatile
class	export	implements	package	this	while
const	extends	import	private	throw	with
continue	false	in	protected	throws	


- *Note : certains de ces mots ne sont pas utilisés dans la version actuelle de Javascript...*

La syntaxe de Javascript

- Les instructions Javascript se terminent par ;.
- Mais ce n'est pas obligatoire... un retour à la ligne peut suffire (à éviter, pour la clarté).
- Ceci dit, **attention** !

```
function produit (x, y) {  
    var prod =  
        x + y  
    return  
        x + y  
}
```

Les valeurs manipulables

- 
- **Types de valeurs**
 - **Littéraux**
 - Comment écrire des valeurs ?
 - **Opérations**
 - **Conversions explicites et implicites**
 - Un sujet délicat !

Ensuite : *instructions en Javascript*

Valeurs : types de valeurs (1/2)

- Obtenir le **type** d'une valeur / variable :
 - `typeof 12` donne **"number"**
 - `typeof true` donne **"boolean"**
 - `typeof "salut"` donne **"string"**
- Autres types :
 - **"function"** : les fonctions
 - **"object"** : tout le reste, y compris les tableaux
 - dont `null` (attention, pas NULL ni Null !)
 - **"undefined"** : variables sans valeur
 - pour les variables déclarées mais pas (encore) initialisées

Valeurs : types de valeurs (2/2)

- JS est un langage **non/faiblement typé**.
 - Pas besoin de donner un type lors de la déclaration d'une variable.
 - Une variable peut changer de type au cours de l'exécution.

```
var x = 7;  
x += " nains";
```
- **Deux implications**
 - Le type d'une variable n'est connu qu'à l'exécution : **pas de vérification de type** !
 - Javascript réalise de nombreuses conversions implicites (avec des règles parfois étonnantes).

Valeurs : littéraux (1/2)

- Littéraux de type **"number"**
 - 42, -78, 342.17, 1.36E78, 037 (octal), 0x3BFF (hexa)
 - tous codés sur 64 bits (sans distinction entier/réel)
 - Valeurs spéciales :
 - Infinity résultat de 5/0
 - -Infinity
 - NaN résultat de 100/"Hello" ("not a number")
 - qu'on peut tester avec isFinite(x) et isNaN(x).
 - Exemple :

```
if (isFinite(a/b)) alert("Division ok !");
```

Valeurs : littéraux (2/2)

- Littéraux de type **"string"**
 - encadrés par des guillemets ou des apostrophes (au choix)
 - Utile : `document.write('<p class="intro">...</p>');`
 - `"pommes", 'chaîne', "aujourd'hui", 'aujourd\'hui'`
 - caractères échappés : `\n \t \' \'\' \"`
 - via le code : `\xA5` ou `\x12B6` (hexa), `\123` (octal)
- Littéraux de type **"boolean"**
 - `true`
 - `false`

Valeurs : opérateurs (1/2)

- Principaux **opérateurs** :
 - sur les nombres : `+` `-` `*` `/` `%` `++` `--`
 - `%` sur les réels : `4.7 % 1.3` donne 0.8
 - concaténation : `+`
 - logiques : `!` `&&` `||` et l'opérateur ternaire `? :`
 - comparaison : `<` `>` `<=` `>=` `==` `!=` `===` `!==`
 - avec conversion implicite : `3 == "3"` donne `true`
 - sans conversion implicite : `3 === "3"` donne `false`
 - opérateurs bit à bit : `&` `|` `~` `^` `<<` `>>` `>>>`
 - opérateur `typeof`

Valeurs : opérateurs (2/2)

- 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>
```

```
<a href="javascript:void(document.form.submit())">  
Envoyer le formulaire</a>
```

Valeurs : conversions (1/12)

- Javascript effectue des conversions de type implicites en fonction des opérateurs.
 - Dans certains cas, le résultat « coule de source ».
 - `"27" * "2"` donne 54 (conversion en nombres)
 - `if (nombre) ...` équivaut à `if (nombre != 0) ...`
 - Mais pas toujours...
 - `"27" + 2` donne "272" !
- D'où l'utilité de connaître l'existence des règles de conversion implicites (*mais pas forcément de les connaître par cœur*) !
 - (1) Comment Javascript convertit-il ?
 - (2) Quand Javascript convertit-il ?

Valeurs : conversions (2/12)

- Comment JS convertit-il... **vers une chaîne de caractères** :
 - réalisable avec `String(...)`

De en "string"
undefined	"undefined"
Objets	null → "null" autres objets : applique <code>.toString()</code>
Booléens	true → "true" false → "false"
Nombres	NaN → "NaN" Infinity → "Infinity" autres nombres → leur écriture

Valeurs : conversions (3/12)

- Comment JS convertit-il... **vers un nombre** :
 - réalisable avec `Number(...)`

De en "number"
undefined	NaN
Objets	null → 0 autres objets → selon <code>.valueOf()</code>
Booléens	true → 1 false → 0
Chaînes	"" (vide ou blancs) → 0 "nombre" → <i>nombre</i> (vide/blancs ignorés) autres chaînes → NaN

Valeurs : conversions (4/12)

- Comment JS convertit-il... **vers un booléen** :
 - réalisable avec `Boolean(...)`

De en "boolean"
<code>undefined</code>	false
Objets	null → false autres objets → true
Nombres	0 et NaN → false autres nombres → true
Chaînes	"" (vide) → false autres chaînes → true

- Valeurs dites "Falsy" : `undefined`, `null`, `0`, `NaN` et `""`

Valeurs : conversions (5/12)

- 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"`

`3 - "2"`

`"15" + "1"`

`7 + true`

Valeurs : conversions (6/12)

- 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, null == undefined renvoie true
 - sinon, les autres cas où null ou undefined apparaît donnent false
 - sinon, tout convertir en nombres

- **Exemples**

`"4" == true`

`5 == "5"`

`'2' == 2`

`"\t\n" == 0`

`0 == ""`

`"" == "0"`

`0 == "0"`

`false == undefined`

`true == null`

`false == null`

Valeurs : conversions (7/12)

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

Valeurs : conversions (8/12)

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

Valeurs : conversions (9/12)

- Sémantique de l'opération logique **A || B**
 - On calcule la valeur de l'expression A, soit a.
 - Si a converti en booléen donne true,
 - le résultat est a.
 - Sinon,
 - le résultat est la valeur de l'expression B.
- Exemple
 - `(12 + 5) || "erreur"`
 - `17 || "erreur"`
 - `17`
 - `"" || 33`
 - `33`

Valeurs : conversions (10/12)

- Pratique pour 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.
- 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;
```


Valeurs : conversions (11/12)

- Sémantique de l'opération logique **A && B**
 - On calcule la valeur de l'expression A, soit a.
 - Si a converti en booléen donne false,
 - le résultat est a.
 - Sinon,
 - le résultat est la valeur de l'expression B.
- Exemple
 - (12 + 5) && "ok"
 - 17 && "ok"
 - "ok"
 - undefined && 33
 - undefined

Valeurs : conversions (12/12)

- Cette particularité est pratique pour 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.

Les instructions en Javascript

- 
- Plutôt standard... !

Ensuite : *fonctions en Javascript*

Instructions : affectations, blocs

- Affectations

```
id = expr      x = 37  
total = prix * quantite  
a = b = c = 4 * x
```

- Ainsi que les raccourcis += -= *= /= %= ++ --

- Blocs d'instructions

```
{ instructions }
```

Instructions : conditionnelles

- Structure conditionnelle simple (if)

```
if (condition)
  instruction
[ else instruction ]
```

- Structure conditionnelle avec gardes (switch)

```
switch (expr)
{
  case val : instruction
  ...
  [ default : instruction ]
}
```

- **break** pour éviter de passer d'un cas à l'autre !

Instructions : itératives

- Boucles `while` et `do`

`while (condition)`
`instruction`

`do`
`instruction`
`while (condition)`

- Boucle `for` (voir aussi `for-in` pour les objets)

`for (init ; condition ; incrémentation)`
`instruction`

- On peut déclarer à la volée dans l'init.
- Utiliser `,` pour séparer les composantes complexes du `for`.

- Échappements


`break`

`continue`

Instructions : conditions

- Les conditions (if, while, ...) sont **converties en booléens** :
 - Tout devient true SAUF
 - pour "number" : 0 et NaN
 - pour "string" : ""
 - undefined
 - null
- Donc, **if (!x)** signifie...
 - si x est null, pas défini, 0, NaN, false ou la chaîne vide
- Donc, **if (x)** signifie...
 - si x est défini en tant que nombre non nul, chaîne non vide, booléen true ou objet (autre que null).

Les fonctions en Javascript

- 
- **Définition de fonctions**
 - **Fonctions anonymes**
 - Littéraux de type « fonction »
 - **Fonctions prédéfinies**
 - **Fonctions et scopes**
 - **Variables locales et variables globales**
 - **Hoisting**

Fonctions

- Déclaration d'une fonction

```
function nom ( ident , ident ... ) {  
    instructions  
}
```

- Quitter la fonction / renvoyer une valeur :

```
return [ expr ]
```

- Appel d'une fonction

```
nom ( expr , expr ... )
```

- Attention : aucune vérification ni de type ni de nombre entre les paramètres formels et les arguments effectifs !

Fonctions anonymes

- Fonctions sous la forme d'expressions (lambda-expressions)
`function [nom] (ident , ident ...) { instructions }`

- Exemple :

```
var double = function (x) { return x * 2; };
```

équivalent (au hissage près) à

```
function double (x) { return x * 2; };
```

- On peut préciser le nom si on le désire. Exemple :

```
var f = function fact(x) {  
    if (x == 0)  
        return 1;  
    else  
        return x * fact(x-1);  
};
```

Fonctions anonymes

- Fonctions sous la forme d'objets

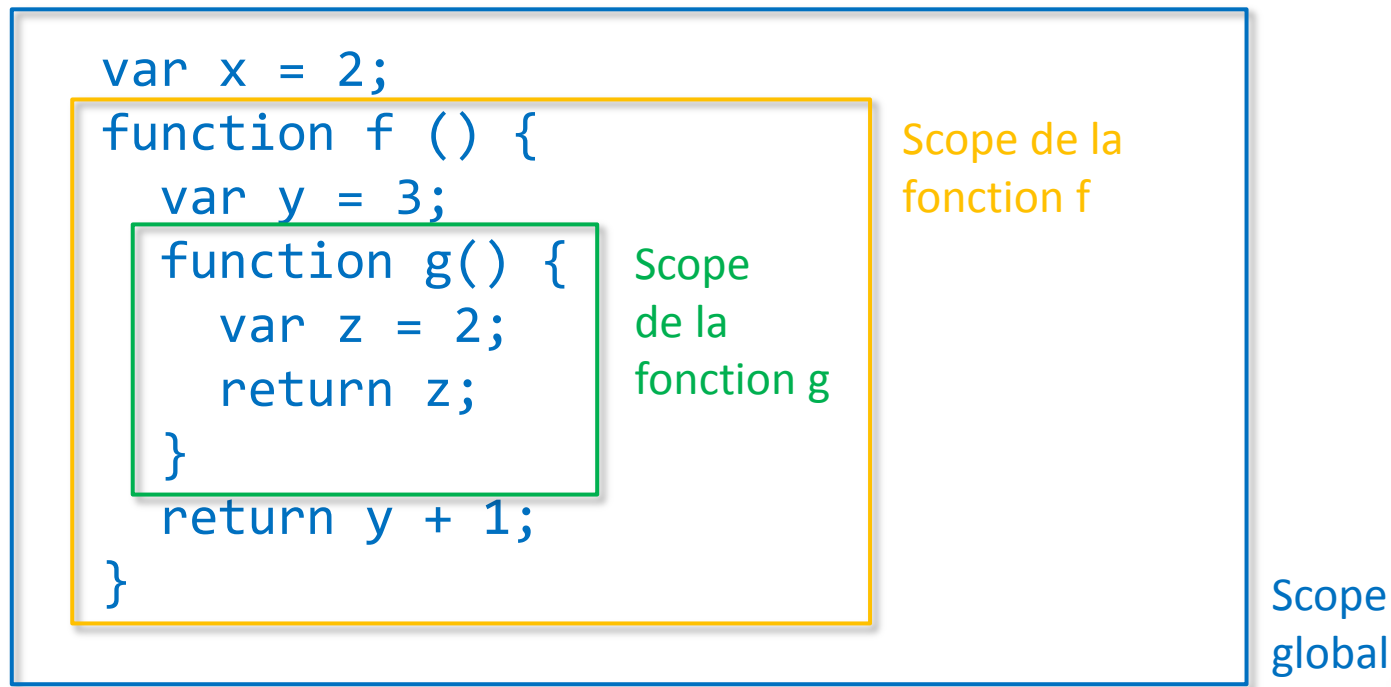
```
new Function ("ident", "ident" ..., "instructions")
```

- Exemple :

```
var double = new Function ("x", "return x * 2;");
```

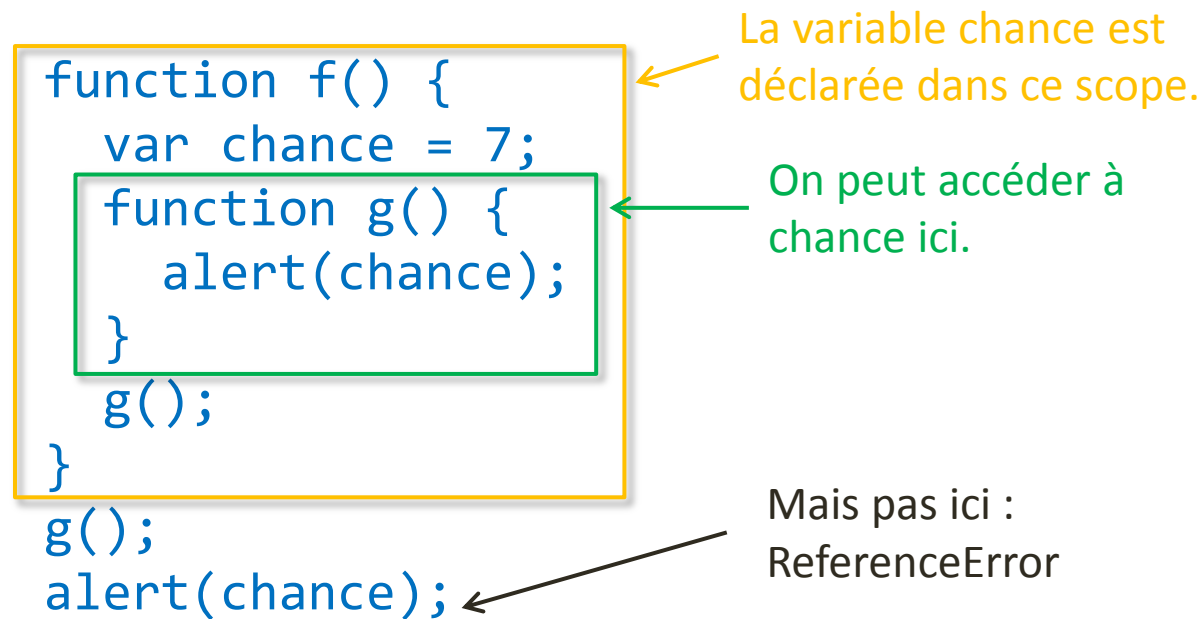
Scope (portée)

- Chaque déclaration de fonction définit un **scope** (portée, contexte), en plus du scope global.



Scope (portée)

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



Scope (portée)

- En Java / C, chaque bloc d'instructions définit un scope. En Javascript, **seules les définitions de fonctions** le font !
- Que vont produire les appels `f(7)` et `f(-3)` ?

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

En Java/C, deux scopes différents.
En Javascript : un seul scope !

Variables locales et globales

- Déclaration d'une variable (avec/sans initialisation)

`var nom`

`var nom = expr`

- Inutile de préciser un type !
- Sans initialisation : la variable est mise à `undefined`.
- Déclarer une constante : `const` au lieu de `var`.
- Une variable ainsi déclarée peut être utilisée
 - dans le scope où elle est déclarée et
 - dans tous les scopes intérieurs.

Variables locales et globales

- Quand l'interpréteur JS rencontre une **utilisation** de la valeur d'une variable (ex : `alert(x)`),
 1. il recherche tout d'abord la variable dans le scope local ;
 2. s'il ne la trouve pas, il passe au scope englobant ;
 3. (et ainsi de suite)
 4. s'il arrive au scope global et qu'il ne trouve toujours pas la variable, il déclenche une **erreur à l'exécution**.
- Concrètement, quand on cherche à utiliser la valeur d'une variable, on cible la déclaration la plus "locale".
- Si aucune variable de ce nom n'est déclarée, on soulève une erreur à l'exécution.

Variables locales et globales

- Quand l'interpréteur JS rencontre une **affectation** concernant une variable (ex : `x = 19`),
 1. il recherche tout d'abord la variable dans le scope local ;
 2. s'il ne la trouve pas, il passe au scope englobant ;
 3. (et ainsi de suite)
 4. s'il arrive au scope global et qu'il ne trouve toujours pas la variable, il la **déclare au niveau global**.
- Concrètement, quand on cherche à donner une valeur à une variable, on cible la déclaration la plus "locale".
- Si aucune variable de ce nom n'est déclarée, on en crée une au niveau global (puis on réalise l'affectation).
- Ceci permet des **déclarations implicites**.

Variables locales et globales

- Exemple : que produit le code suivant ?

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

Variables locales et globales

- 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, les déclarations (de variables et de fonctions) sont **hissées** vers le début.

- Exemple :

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



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

- Concrètement, cela signifie qu'on peut utiliser une fonction même si sa définition se trouve plus loin dans le même scope (ou plus loin dans le même élément <script>).

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);  
  var x = 3;  
}  
f();
```



Seule la déclaration est hissée !
Pas l'initialisation

Exercice (1/5)

- Que va afficher le script suivant ?

```
var x = 7;  
function f () {  
    var x = 2;  
    x++;  
    console.log(x);  
}
```

```
console.log(x);  
f();  
console.log(x);
```

Exercice (2/5)

- Que va afficher le script suivant ?

```
var x = 7;  
function f () {  
  var x = 2;  
  x++;  
  console.log (x);  
}
```

```
console.log(x);  
f();  
console.log(x);
```


Exercice (3/5)

- Que va afficher le script suivant ?

```
function f () {  
    var x = 5;  
    if (x == 5) {  
        var x = 7;  
        console.log (x);  
    }  
    console.log (x);  
}  
  
f();
```

Exercice (4/5)

- Que va afficher le script suivant ?

```
function f () {  
    var x = 11;  
    function g () {  
        console.log (x);  
        var x = 13;  
    }  
    g();  
    console.log (x);  
}  
  
f();
```

Exercice (5/5)

- Que va afficher le script suivant ?

```
var x = 17;  
function f (z) {  
    if (z)  
        var x = 4;  
    console.log (x);  
}
```

```
f(true);  
f(false);
```