

Module 3B

Fonctions, tableaux et objets

Technologies web
HENALLUX — IG2

Fonctions, tableaux et objets

➤ Les fonctions nommées

- Fonctions « habituelles » comme en C et en Java
- *(plus tard : fonctions anonymes)*


➤ Les tableaux

- Hétérogènes et dynamiques

➤ Les objets

- Première approche des objets en tant que structures

Les fonctions nommées

- 
- **Définir une fonction nommée**
 - Trois manières de procéder
 - **Utiliser une fonction nommée**
 - Paramètres formels et paramètres effectifs
 - **Quelques fonctions prédéfinies**
 - **Fonctions : un exemple**

Ensuite : *Les tableaux*

Définir une fonction nommée

- Méthode 1 : **déclaration standard**

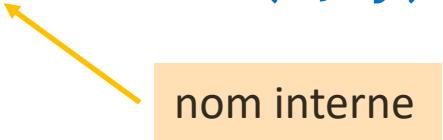
```
function affiche (x) {  
    console.log("La valeur est : " + x);  
}  
  
function produit (x, y) {  
    return x * y;  
}
```

- Syntaxe similaire au C et au Java (mais sans les types !)
- « **return** » et « **return expr** » possèdent la signification habituelle.
- La déclaration est « **hoistée** » en tête de scope fonctionnel.

Définir une fonction nommée

- Méthode 2 : **expression fonctionnelle**

```
let affiche = function (x) {  
    console.log("La valeur est : " + x);  
};  
const produit = function calculeFois (x, y) {  
    return x * y;  
};
```



nom interne

- Il s'agit d'une déclaration de variable.
- Avec **var**, la déclaration est « **hoistée** » en tête de scope fonctionnel mais pas l'initialisation.
- Avec **let/const**, le principe de la TDZ s'applique.
- Le nom « interne » est facultatif.

Définir une fonction nommée


- Méthode 3 : **via le constructeur Function**

```
let affiche = new Function ("x",  
    'console.log("La valeur est : " + x);');  
const produit = new Function ("x", "y",  
    'return x * y;');
```


- Une fonction est un objet.
- Arguments du constructeur : les **paramètres formels** puis, en dernier, le **code** (tout sous forme de chaînes de caractères).
- Avec **var**, la déclaration est « hoistée » en tête de scope fonctionnel mais pas l'initialisation.
- Avec **let/const**, le principe de la TDZ s'applique.
- [Clean Code] **à éviter**

Définir une fonction nommée

- Définitions de fonctions et hoisting



```
function test () {  
  console.log(f);  
  console.log(f(3));  
  function f(x) {return x;}  
}  
test();  
// function f()  
// 3
```



```
function test () {  
  console.log(f);  
  console.log(f(3));  
  var f = function(x) {return x;};  
}  
test();  
// undefined  
// TypeError: f is not a function
```

- Et si on avait utilisé "let" ou "const" au lieu de "var" ?
// ReferenceError: can't access lexical declaration 'f'
before initialization

Utiliser une fonction nommée

- Appeler une fonction nommée

```
function affiche (x) { ... }  
function produit (x,y) { return x * y; }  
affiche(17 + 25);  
let prod = produit(a,b);
```

paramètres formels

paramètres effectifs

- **Attention !** Aucune vérification de type !
- **Aucune vérification de nombre non plus !**
 - Trop de paramètres effectifs : les derniers sont ignorés.
 - Pas assez de paramètres effectifs : complété avec des « undefined ».
- Passage par valeur pour les types primitifs
 - **Attention !** Les chaînes de caractères sont des valeurs primitives !
 - Passage par référence pour les objets (dont array, fonctions, ...)

Quelques fonctions prédéfinies

- Quelques exemples :
 - `eval` retourne la valeur d'un code Javascript (! plutôt lent)
`eval("isNaN(34)");`
`eval("alert('Salut !'); var x = 17;");`
 - `isNaN`, `isFinite`
- Fonctions de conversion forcée :
 - `parseInt(s)` ou `parseInt(s, base)` : string vers entier
 - `parseFloat(s)` : string vers nombre réel
 - `Number(obj)` : convertit en nombre
 - `String(obj)` : convertit en chaîne de caractères
 - `Boolean(obj)` : convertit en booléen

Exercice (examen janvier 2016)

1. On considère le code ci-dessous. Qu'affichera la console ?

```
// Définition 1
function f(x) { console.log(x+7); }
// Appel 1
f(2);
// Définition 2
function f(x) { console.log(x+3); }
// Appel 2
f(3);
```



5
6

2. Si, dans la question précédente, on écrit les deux définitions de fonctions sous la forme

```
var f = function (x) { ... };
```


qu'affichera la console ?

9
6

3. Les réponses aux deux questions précédentes sont différentes. C'est dû à un principe qui, en Javascript, s'applique un peu différemment aux définitions de fonctions et aux déclarations/initialisations de variables. Quel est le nom de ce principe ?

hoisting

Les tableaux

- 
- **Caractéristiques des tableaux Javascript**
 - hétérogènes, dynamiques et peuvent contenir des trous
 - **Créer un tableau**
 - **Utiliser un tableau**
 - **Tableaux à trous**
 - **Parcourir un tableau**

Ces encadrés indiquent des éléments plus avancés.

Ensuite : *les objets*

Caractéristiques des tableaux

En Javascript, les **tableaux**...

- sont **hétérogènes** (et potentiellement **multidimensionnels**)

```
let tab = [34, true, 'bonjour !', [1, true, "three"]];
```

Valeur : Array [34, true, 12.34, Array[3]]

34	True	12.34	[...]
----	------	-------	-------

- sont **dynamiques** (leur taille peut évoluer)

```
tab[4] = 'nouv';           ou tab.push('nouv');
```

34	True	12.34	[...]	"nouv"
----	------	-------	-------	--------

- peuvent contenir **des trous**

```
tab[7] = -72;
```

Valeur : Array [34, true, 12.34, Array[3], "nouv", <2 empty slots>, -72]

34	True	12.34	[...]	"nouv"			-72
----	------	-------	-------	--------	--	--	-----

Créer un tableau

- Deux méthodes pour **créer un tableau** :

- Utiliser un **littéral**

```
let tab = [1,2,3];
```

- Utiliser le **constructeur** new Array

```
let tab = new Array (1,2,3);
```

1	2	3
---	---	---

- Cas 1 : **Tableau vide** (longueur = 0)

```
let tabVide = [];
```

```
let tabVide = new Array ();
```

Affichage : Array []

- Cas 2 : **Tableau de *n* cases** (trous)

```
let tab5Trous = new Array (5);
```

```
let tab5Trous = [,,, ,];
```

Affichage : Array [<5 empty slots>]

--	--	--	--	--

Créer un tableau

- Cas 3 : **Tableau avec valeurs**

```
let jours = ["lun", "mar", "mer", "jeu", "ven"];  
let jours = new Array("lun", "mar", "mer", "jeu", "ven");  
let tabValeurs = new Array (1, 2, 3, 4, 6, 12);  
let tab = [1, "2", true, function () { return 42; }];  
let tab2D = [[1, 2], [3, 4, 5]];
```

```
let monTab = [0, 1, 2, 3,]; // dernière virgule optionnelle
```

```
let tabTrous = [0, , , 3]; // avec trous
```

0			3
---	--	--	---

- Attention !**

```
new Array("3") → Array [ "3" ]
```

```
new Array(3) → Array [ <3 empty slots> ]
```

```
[3] → Array [ 3 ]
```

Utiliser un tableau

- **Accéder aux éléments** (lecture ou écriture)

```
jours[2] = "Wednesday";  
for (let i = 0 ; i < 5 ; i++) console.log(jours[i]);  
matrice[1][2]
```

- Accès hors borne (indice \geq longueur) :
 - En lecture : renvoie undefined
 - En écriture : on modifie la longueur du tableau !
- **Attention !** Le tableau doit être déclaré avant de pouvoir y écrire une valeur : pas de `t[4] = 3` sans `t = []` ou autre déclaration préalable

- **Tester les indices utilisés : in**

```
let tabT = [0, 1, , , 4];  
1 in tabT → true // tabTrous[1] existe  
3 in tabT → false // trou en pos 3, tabT[3] donne undefined  
6 in tabT → false // 6 > longueur, tabT[6] donne undefined
```

0	1			4
---	---	--	--	---

Utiliser un tableau

- **Taille d'un tableau** (lecture ou écriture)
 - Pour obtenir la taille d'un tableau :
`tab.length` (on peut aussi écrire `tab["length"]`)
- Il s'agit d'une **propriété modifiable** (taille dynamique) !

```
let tab = new Array (1, 2, 3, 4, 5);
console.log(tab.length);           // 5
console.log(tab[3]);               // 4
tab[8] = 6;
console.log(tab.length);           // 9
tab.length = 3;
console.log(tab[3]);               // undefined
tab.length = 6;
console.log(tab[3]);               // undefined
```

 - Quand on diminue la longueur, on supprime les cases hors-limite (leurs valeurs sont perdues définitivement).
 - Quand on augmente la longueur, on ajoute des cases vides (trous).

Tableaux à trous (à éviter)

- **Tableau à trous** = tableau dont certaines cases n'existent pas.

```
let tabTrous = new Array (6);
tabTrous[0] = 1; tabTrous[1] = 2; tabTrous[4] = 5;
```

```
let tabTrous = [1, 2, , , 5, ,]; // dernière virgule ignorée
```

1	2			5	
---	---	--	--	---	--

- On peut voir les tableaux comme une série d'associations $i \rightarrow tab[i]$.
Ici : $0 \rightarrow 1$; $1 \rightarrow 2$; $4 \rightarrow 5$
- Trou = association manquante.

- **Attention** : Trou \neq case contenant undefined

```
let tab = [1, 2, undefined, undefined, 5, undefined];
0  $\rightarrow$  1 ; 1  $\rightarrow$  2 ; 2  $\rightarrow$  undefined ; 3  $\rightarrow$  undefined ; 4  $\rightarrow$  5 ; 5  $\rightarrow$  undefined
```

```
3 in tab donne false
```

```
3 in tab donne true
```

1	2	udf	udf	5	udf
---	---	-----	-----	---	-----

Parcourir un tableau

- **Trois types de boucles**

- Boucles usuelles : for, while, do ... while
- Boucles for-in : parcourir les indices des associations
- Boucles for-of : parcourir les valeurs des associations

- **Boucle usuelle** (for, while, do ... while)

```
for (let i = 0 ; i < tab.length ; i++)  
  { console.log(tab[i]); }
```

- « let » permet d'avoir une variable locale à la boucle (avec var : scope fonctionnel)
- On peut également utiliser une variable qui existe déjà.
- *Note : avec « let », une nouvelle variable i est créée à chaque itération et initialisée à la valeur précédente (note importante pour la notion de closure, voir plus tard).*

Parcourir un tableau

- **Boucle « for in »**

```
for (let i in tab) console.log(tab[i]);
```

- i prend comme valeurs les indices des cases du tableau.
- « const » également possible (on déclare à chaque itération).

- Les trous sont sautés/ignorés.

- **Boucle « for of »**

```
for (let val of tab) { console.log(val); }
```

- val prend comme valeurs le contenu des cases du tableau.
- « const » également possible (on déclare à chaque itération).

- Les trous ne sont pas sautés/ignorés mais considérés comme contenant undefined !


Parcourir un tableau

- Exemple

```
let tab = [3, "hello", , true];
let sorties = ["", "", ""];
for (let i = 0 ; i < tab.length ; i++)
    sorties[0] += i + "->" + tab[i] + ", ";
for (let i in tab)
    sorties[1] += i + "->" + tab[i] + ", ";
for (let val of tab)
    sorties[2] += val + ", ";
console.log(sorties[0]);
// 0->3, 1->hello, 2->undefined, 3->>true,
console.log(sorties[1]);
// 0->3, 1->hello, 3->true,
console.log(sorties[2]);
// 3, hello, undefined, true,
```

3	Hello		true
---	-------	--	------

Les objets (niveau 1)

- 
- **Le concept de « tableau associatif »**
 - hétérogènes, dynamiques et peuvent contenir des trous
 - **Les objets en Javascript**
 - **Créer un objet**
 - **Utiliser un objet**
 - **Valeurs primitives et objets**
 - **L'orienté objet en Javascript**

Objets et « niveaux »

Pour faciliter l'approche des objets en JS, on peut découper les concepts en plusieurs niveaux :

- Niveau 1 : les **objets**
 - Objet = tableau associatif dynamique
- Niveau 2 : les **prototypes**
 - Prototype = objet où on stocke des propriétés communes à plusieurs objets
- Niveau 3 : les **constructeurs**
 - Constructeur = fonction qui crée des objets selon un « moule »
- Niveau 4 : l'**héritage**, ancienne méthode
 - Assez complexe... mécanique « de bas niveau »
- Niveau 5 : la nouvelle **syntaxe** (ES6)
 - Syntaxe qui « cache » la mécanique se trouvant « sous le capot »

Tableaux associatifs

- **Tableaux standards** / à index numérique :

Point de vue « logique »

- Tableau = suite de "cases" contenant des valeurs
- Cases numérotées (la plupart du temps) 0, 1, 2, 3...
- Valeurs accessibles via `tab[0]`, `tab[1]`, `tab[2]`...

La manière dont c'est implémenté en Javascript...

- Tableau = groupe d'associations du type : indice → valeur
`let tab = ["oui", true, -42];`
0 → "oui" ; 1 → true ; 2 → -42
- Les indices correspondent aux premiers naturels.
(En fait, il s'agit de strings contenant les premiers naturels.)

- **Tableaux associatifs**

- Les "cases" ne sont plus numérotées mais repérées par des "clefs" quelconques (souvent alphanumériques...).

Tableaux associatifs

- Un **tableau associatif** (ou **dictionnaire**, **map**, **dictionary**) est une structure qui associe des **valeurs** à certaines **clefs**.
 - Dictionnaire : associe des valeurs (= définitions) à des mots (= clefs).
 - Les **clefs** sont (souvent) des chaînes de caractères.
 - Les **valeurs** peuvent être de n'importe quel type.
- **Exemples**
 - `année["HTML"] = 1` clef = "HTML", valeur = 1
 - `année["CSS"] = 1` clef = "CSS", valeur = 1
 - `année["Javascript"] = 2` clef = "Javascript", valeur = 2
 - `année["PHP"] = 3` clef = "PHP", valeur = 3

 - `traduction["dog"] = "chien"`
 - `traduction["brother"] = "frère"`
 - `traduction["end"] = "fin"`

Tableaux associatifs

- Le concept de **tableau associatif** est une notion « logique » qu'on retrouve sous diverses formes dans la plupart des langages :
 - **En C** : structures (version limitée)
 - **En Java** : classe Map (et ses filles)
 - **En Javascript** : objet = tableau associatif
 - *Attribut* : association clef → valeur (nombre, entier, réf d'objet, ...)
 - *Méthode* : association clef → fonction
- Autre exemple pour s'ouvrir l'esprit...
 - `tab["doubler"] = function (x) { return x * 2; }`
 - `tab["mettre au carré"] = function (x) { return x * x; }`
 - `tab["incrémenter"] = function (x) { return x + 1; }`

Objets en Javascript

- En Javascript : **objet = tableau associatif**
 - **Clefs** = noms des propriétés (attributs ou méthodes)
 - Pour les attributs : **valeur** = valeur de l'attribut
 - Pour les méthodes : **valeur** = fonction décrivant la méthode

Clef	Valeur
prenom	"Homer"
nom	"Simpson"
parle	<pre>function () { alert("Doh !"); }</pre>
toString	<pre>function () { return this.prenom + " " + this.nom; }</pre>

Mot spécial « this »

- Signification complexe
- En 1^{re} approche : référence à l'objet sur lequel on exécute la méthode

Objets en Javascript

- De plus, les objets Javascript sont **dynamiques** :
 - Des **attributs** peuvent être ajoutés/supprimés/modifiés à la volée.
 - Des **méthodes** peuvent être ajoutées/supprimées/modifiées à la volée.
- Différence d'approches entre
 - **OO par classes** et
 - Classe = moule
 - Objets = instances d'une classe
 - **OO (prototypal)** de Javascript !
 - Objets = entités à part entière, peuvent exister sans « modèle »

Créer un objet

- **Comment créer un objet en Javascript ?**
 - En Java, il faut tout d'abord définir une classe puis l'instancier.
 - En Javascript, on peut définir un objet directement, sans le lier à une classe !
- Deux méthodes pour créer un objet :
 - Créer un **objet vide**, puis lui **ajouter** des propriétés.
 - Similaire aux tableaux :
`t = []; t[0] = 17; t[1] = -5; ...`
 - Créer un **objet qui possède déjà des propriétés** avec un littéral.
 - Similaire aux tableaux :
`t = [17, -5, ...];`
 - (Il existe d'autres méthodes.)

Créer un objet (méthode 1)

- Création d'un **objet vide**

```
let h = {};
```

- **Ajout de propriétés**

- Syntaxe des tableaux associatifs

```
h["prenom"] = "Homer";  
h["nom"] = "Simpson";  
h["parle"] = function () { alert("Doh !"); };  
h["toString"] = function ()  
  { return this.prenom + " " + this.nom; };
```

- Syntaxe orienté objet

```
h.prenom = "Homer";  
h.nom = "Simpson";  
h.parle = function () { alert("Doh !"); };  
h.toString = function ()  
  { return this.prenom + " " + this.nom; };
```

Créer un objet

- Les **noms des propriétés** peuvent être n'importe quelles chaînes de caractères.
- **Pas limité** par les règles sur les identificateurs (syntaxe, mots réservés...)

```
h["for"]=17;    ou h.for = 17;
```

```
h["6po"]=true;  
h["date naissance"] = 20001112;
```

La syntaxe avec "." n'est pas utilisable quand ça ne correspond pas à la syntaxe des identificateurs.

- On peut également utiliser des **nombres** autorisés (qui sont automatiquement convertis en chaînes).

```
h[4] = "quatre";  
h["4"] = "quatre";    // identique au précédent  
(pas de syntaxe avec point)
```

Créer un objet (méthode 2)

- Création via un **littéral pour objet**

```
const h = {  
  "prenom" : "Homer",  
  "nom" : "Simpson",  
  "parle" : function () { alert("Doh !"); },  
  "toString" : function () {  
    return this.prenom + " " + this.nom;  
  }  
};
```


- Association au format "**clef**" : **valeur** séparées par des virgules
- Plusieurs options de « sucre syntaxique » (voir juste après)
- Syntaxe à la base du format **JSON** (voir suite)
 - JSON = JavaScript Object Notation

Créer un objet (méthode 2)

Cinq remarques sur la syntaxe des littéraux pour objets

- (1) On peut souvent **omettre les guillemets** autour des clefs/noms de propriété (exception : quand le nom est formé de plusieurs mots).

```
const h = {  
  prenom : "Homer",  
  nom : "Simpson",  
  parle : function () { alert("Doh !"); },  
  toString : function ()  
    { return this.prenom + " " + this.nom; },  
};
```



- (2) On peut ajouter une virgule à la fin de la liste des propriétés.

Créer un objet (méthode 2)

- (3) [ES6] On peut utiliser la **syntaxe courte pour les méthodes**

```
const h = {  
  prenom : "Homer",  
  nom : "Simpson",  
  parle () { alert("Doh !"); },  
  toString () { return this.prenom + " " + this.nom; },  
};
```

- (4) Définition de **propriétés raccourcie en utilisant une variable**

```
function sommeEtProduit (x,y) {  
  let somme = x + y;  
  let produit = x * y;  
  // return { somme : somme, produit : produit };  
  return { somme, produit };  
}  
let res = sommeEtProduit (10,7);  
console.log(res); // Object { somme: 17, produit: 70 }
```

Créer un objet (méthode 2)

- (5) **Noms de propriétés calculés**

```
function infoArticle (nom, prixHtvaEU, veutDollars) {  
  const tva = 0.21;  
  const tauxEuDo = 1.12;  
  const devise = veutDollars ? "DO" : "EU";  
  const prixHtva = prixHtvaEU * (veutDollars ? tauxEuDo : 1);  
  const prix = prixHtva * (1 + tva);  
  return { nom, ["prix" + devise] : prix };  
}
```

```
infoArticle("art1", 10, false);  
→ { nom: "art1", prixEU: 12.1 }
```

```
infoArticle("art2", 10, true);  
→ { nom: "art2", prixDO: 13.552000000000001 }
```

Utiliser les objets

- **Accéder aux propriétés** (en lecture ou en écriture)
 - Deux syntaxes possibles : `h.nom` ou `h["nom"]`
 - On peut modifier (ajouter/modifier) les propriétés

```
alert(h["nom"]);           // en lecture  
console.log(h.prenom);
```

```
h["prenom"] = "Marge";    // en écriture  
h.nom = "Shepard";
```

```
h["parle"]();             // appel  
h.parle();  
alert(h["toString"]());  
alert(h.toString());
```

Utiliser les objets

- **Modifier les propriétés**

- Les objets Javascript sont **dynamiques** : on peut...

- **ajouter** de nouvelles propriétés (attributs ou méthodes)

```
h.nourriture = "donuts";
```

- **modifier** une propriété existante (attribut ou méthode)

```
h.toString = function () {  
    return this.nom + ", " + this.prenom;  
};
```

Note : une propriété peut être déclarée immuable ("readonly"). Dans ce cas-là, quand on tente de la modifier, rien ne se passe.

- **supprimer** une propriété existante (attribut ou méthode)

```
delete h.parle;
```

- Un objet créé avec const peut voir son contenu modifié !

```
const o = { nom : "mon objet" };  
o.nom = "autre nom";           // on peut modifier l'attribut  
o = { nom : "nouvel objet" };  // Erreur : mais pas la référence !
```

Utiliser les objets

- **Test d'égalité** entre deux objets
 - Correspond à un test d'égalité de pointeurs/références !
 - Même sémantique pour `==` et pour `===`

```
let obj1 = {};  
obj1.valeur = 237;
```

```
let obj2 = {};  
obj2.valeur = obj1.valeur;
```

```
obj1 == obj2 → false
```

```
obj1 === obj2 → false
```

Utiliser les objets

- L'opérateur **in** : `"ident" in obj`
 - indique si l'objet possède une propriété du nom indiqué
- Exemples :
 - `"nom" in h → true`
 - `"travail" in h → false`

Utiliser les objets

- La boucle **for-in** : `for (ident in obj) instr`
 - passe en revue toutes les propriétés (énumérables) de l'objet
 - tour à tour, *ident* prend comme valeur le nom de chacune des propriétés
- Exemple :

```
let msg = "";
for (let prop in h)
  msg += prop + " -> " + h[prop] + "\n";
alert(msg);
```
- *Note : une propriété peut être déclarée "non énumérable" ! Dans ce cas-là, elle ne sera pas visitée par un for-in.*

Utiliser les objets

- La boucle **for-of** : `for (val of obj) instr`
 - passe en revue les valeurs des propriétés (énumérables) de l'objet
 - uniquement pour les **objets "itérables"**
 - = tableaux, chaînes de caractères, maps, sets
 - = la plupart des éléments calculés ressemblant à des tableaux (voir DOM)
- Exemple :

```
for (let lettre of "Hello") console.log(lettre);  
// affiche H e l l o (une lettre par ligne)
```
- *Note : on peut rendre n'importe quel objet itérable en lui ajoutant une propriété indiquant comment visiter ses propriétés énumérables (cela dépasse le cadre du cours – mot-clef : `Symbol.iterator`)*

Utiliser les objets

- [v14, 06/2020] **Nullish coalescing operator** : `expr1 ?? expr2`
 - vaut a priori `expr1`
 - sauf si c'est `null` ou `undefined`, auquel cas, ça vaut `expr2`.

- Exemples :

```
afficher(titre ?? "Sans titre");
```

```
function afficheCri (nomChien) {  
  let nom = nomChien ?? "Un chien";  
  afficher(`${nom} aboie.`);  
}
```

Est-ce la même chose que

```
nom = nomChien || "Un chien" ?
```

Utiliser les objets

- [v14, 06/2020] **Optional chaining operator** : `obj.prop?.sprop`
 - vaut a priori `obj.prop.sprop`
 - sauf si `obj.prop` est `null` ou `undefined`, auquel cas, ça vaut `undefined`.

- Exemples :

```
let bart = {  
  nom : "Bart",  
  chien : { nom : "PetitPapaNoël" }  
};
```

`bart.chien.nom` → "PetitPapaNoël"

`bart.chat.nom` → `TypeError Exception`

`bart.chat` → `undefined`

`bart.chat?.nom` → `undefined`

Utiliser les objets

- [v14, 06/2020] **Optional chaining operator**
 - peut aussi s'utiliser avec l'écriture « tableau associatif »
`bart.chat["n"+"om"] → TypeError Exception`
`bart.chat?.["n"+"om"] → undefined`
 - peut aussi s'utiliser avec des méthodes dont on n'est pas certain de l'existence
`bart.aboie() → TypeError Exception`
`bart.aboie?().() → undefined`

Valeurs primitives et objets

- Comparaison entre les **valeurs primitives** et les **objets**

	Valeurs primitives	Objets
Catégories	nombre booléens chaînes de caractères	fonctions tableaux autres objets
Mutabilité	immuables	mutables (a priori)
Comparaison ==	par valeur	par référence
Passage (argument)	par valeur	par référence

- Immuable = on ne peut pas modifier son état (au sens OO)

```
let s = "debut";  
s[3] = "a"; s.nvProp = -5;           // s inchangé
```

Un point sur l'OO en JS

- Le point jusqu'ici...
 - **Objets** = tableaux associatifs
 - **Méthodes** = propriétés dont la valeur est une fonction (**this** pour faire référence à l'objet)
 - 3 **syntaxes** disponibles : OO (o.prop), tableau (o["prop"]), littéral
 - Structure **dynamique** : on peut ajouter, modifier ou supprimer des propriétés !
 - Pour les objets : **in**, **for in**