chapitre 1 : Les variables

Le Javascript est un langage non type, faiblement typé, typé dynamiquement. Ce qui signifie que le contenue d'une variable peut changer au cours de son exécution.

le nom d'une variable est sensible à la case.

1.1 Les Types

number, ***String***, ***boolean***, ***undefined***. Il existe aussi ***function*** et ***object***.

Il est possible de trouver le type d'une variable au moyen de la fonction **typeof (*variable*)** qui retourne le type de la variable placé en paramètre.

1.1.1 Nombres

- o.... : octal
- ox... : hexa
- 0o... : octal [ES6]
- 0b... : binaire

> util : isFinite(x)
> util (bis) : isNaN(x)

1.1.1.1 Fonctions

Attention, dans les fonctions suivantes, si x n'est pas un nombre, une conversion implicite sera opérée.

```
isFinite(x); // retrourne true si le nombre x est un numéro standard
isNan(x); // retourne true si x est un NaN
```

Strings

Un string peut s'écrire entouré de " ou de '. les ' sont evalués (ex : '${variable}')

> Les strings en Javascript sont immuables. ce qui signifie que tout changement entraine la création d'une novelle chaîne de caractère.

> on peut aussi faire des calculs dans le ${}

Les caractères échappées en JS sont :

```
\n \t \' \" \\.
```

Apostrophe inverse (ES6)

Quand on veut : un texte sur plusieurs lignes ou contenant des expressions à évaluer on peut utiliser les apostrophes inverses.

La norme ES6 permets d'effectuer des opérations dans les gabarits

```
let string = `je suis ${nom} et dans un an, j'aurais ${age+1}`
```

Opérations sur les Strings

- **Concaténation**: str + str
- **Longueur**: str.length
- **Extraction**: str[numLettre] ou str.charAt(numLettre)
- **Test**: str.startWith(str)
- **Test**: str.endsWith(str)
- **Test**: str.includes(str)
- **Extraction**: str.substring(début,longueur)
- **Extraction**: str.substring(debut,fin)
- **Recherche**: str.indexOf(str)
- **Recherche**: str.lastIndexOf(str)
- **Décomposition**: str.split(séparateur)

Booléen

Une variable Booléenne peut prendre 2 valeurs ***true*** et ***false***.

Les valeurs Falsy

Les valeurs dites falsy sont des valeurs qui une fois converties en Booléen donneront false.

- false
- undefined
- null
- 0
- NaN
- ""

> attention "False" n'est pas falsy car une chaine de caractère qui n'est pas null est considérée comme true lors d'une évaluation Booléenne

Date (bonus)

Date est un objet qui est utilisé pour manipuler facilement des dates.

```
let maintenant = new Date();
let minutes = maintenant.getMinutes();
let heures = maintenant.getHours();
```

1.2 La Déclaration d'une variable

Il existe 3 Méthode de déclaration de variable : Let , var et pas déclarée

> contrairement à d'autres langages de programmation, les variables déclarées dans la fonction parent sont aussi disponibles dans la fonction enfant imbriquée dans la première.

> Dans le cas où une variable serait utilisée dans avoir été déclarée, cela provoquerait un RéférenceError

1.2.1 Let

Let est la méthode à préférer si l'on souhaite déclarer une variable.

Les variables ainsi déclarées sont des variables locales au bloc.

exemples :

```
let NOM_VARIABLE = 1234;
let maFonction = function(x,y){ return x+y };
let monNom = "Antoine";
```

1.2.2 Var

Les variables déclarées avec var sont valables dans le scope de la fonction et sont déclarées (mais pas initialisées) au début de celle-ci (cfr explication hoisting/hissage).

Dans le cas d'une deuxième déclaration, la variable sera modifiée sans pour autant être redéclarée.

> En Pratique, on va éviter de les utiliser car ses particularités nuisent à la lisibilité du code et peuvent provoquer des résultats inattendus.

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

1.2.3 Pas de déclaration

Si la variable est initialisée sans déclaration, elle est déclarée au niveau global.

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

1.2.4 Constantes

constante locale au scope du bloc.

Elle **doit être initialisée** lors de sa déclaration.

Il faut préférer utiliser const si la valeur ne change pas

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

Le hissage (= hoisting)

En javascript, les définitions des **fonctions** et des **variables déclarées avec var** sont hissées au début. Attention, car seul la déclaration est hissée et non l'initialisation des variables.

let et **const** voient aussi leur déclaration hissées. mais *au début du bloc*. la zone entre la déclaration et l'initialisation est appelée **TDZ** Temporal Dead Zone.

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

Conversions

La conversion de type permets de changer le type d'une variable. Ces conversions sont aussi effectuées de manière implicites par le programme lui-même

Attention, le langage javascript effectue beaucoup de conversions implicites. Pour le bien de la lecture du code, il est conseillé d'éviter d'effectuer ces conversions implicitement et de les faire explicitement.

```
Number(x)
String(x)
Boolean(x)
parseFloat(x)
```

Type départ ↓ / Type destination →	**Nombre**	**Booléen**	**String**
Nombre	inchangé	0 et Nan → false / autres→ true	NaN→ "NaN" / Infinity → "Infinity" / autre → leur écriture
Booléen	true→ 1 / false → 0	inchangé	true→"true" / false → "false"
String	blanc → 0 / "nombre"→ nombre / "autre" -> NaN	""→ false / reste → true	inchangé
Objet	null → 0 / autreObjet → autreObjet.valueOf()	objet null → false / autre objet → true	objet null → "null" / autre → objet.toString()
Undefined	Nan	false	"undefined"

Conversions Explicites

Pour convertir un string vers un nombre on utilise parseFloat

```
parseFloat(s);
```

> On arrête la conversion au premier caractère illégal rencontré

Pour convertir un string ***s*** en un entier dans une base ***b***

```
parseInt(s,b);
```

> On arrête la conversion au premier caractère illégal rencontré

Les Conversions Implicites

Beaucoup moins visibles que les conversions explicites. Elles sont moins évidentes et plus difficiles que les conversions explicites. Les règles de *Clean Code* nous demandent donc de les éviter.

C'est pourquoi il est conseillé d'utiliser ''===" plustot que '==='.

Les deux expressions suivantes sont donc équivalentes

```
VAL === 123 ;
NUMBER(VAL) == 123;
```

Le '+'

- **Si il y a au moins un string** : Les autres membres seront convertis en string et concaténé.
- **sinon** : on convertira les membres en nombres et on les additionnera

Le "=="

- Mêmes types : Comparaison
  - NaN N'est égal à personne, même pas lui même
- Null == undefined

Le "==="

False si les 2 types sont différents ou si les 2 éléments comparées sont différents.

Il possède l'avantage de ne pas convertir automatiquement les variables.

Le '<'

- Lors d'une comparaison de **2 strings**, il effectuera une comparaison lexicographique.
- **Sinon** il convertira les 2 variables en nombres et effectuera une comparaison

Le '||'

Dans cet exemple, on utilisera l'exemple des valeurs ***a*** et ***b***.

A||B représente cette comparaison.

> A = Bool(a) a est automatiquement convertis en A. (idem pour B)

Si ***A*** est ***true*** alors le résultat de l'opération est a. Sinon, le résultat est b.

On utilise cette propriété afin de définir des valeurs par défaut

Le '&&'

On se base sur le même principe que pour le '||' sauf que ici si A est faux on choisira a et si il est vrai on prendra b.

On utilise cette propriété pour vérifier qu'un objet existe et si c'est le cas on évalue ensuite sa propriété.

exemple :

```
objet && objet.propriété
```