

Les types de données

Table des matières

I. Contexte	3
II. Types numériques	3
A. Les types numériques	3
B. Exercice	5
III. Les autres types primitifs	6
A. Les autres types primitifs	6
B. Exercice	8
IV. Essentiel	8
V. Auto-évaluation	9
A. Exercice	9
B. Test	9
Solutions des exercices	10

I. Contexte

Durée : 45 minutes

Contexte

En programmation, toute donnée a un type. Un type va définir la nature des différentes valeurs que peut prendre une donnée. En abordant les variables, on s'est rendu compte que les données peuvent être de type nombre, de type chaîne de caractères, de type booléen, etc.

Il faut savoir que JavaScript est un langage très peu typé. En effet, il existe des langages comme Java qui sont fortement typés, c'est-à-dire qu'il va falloir, par exemple, indiquer quel est le type d'une variable à chaque fois que l'on déclarera celle-ci. On indiquera alors si c'est un booléen, une chaîne, un entier, etc. À l'inverse, JavaScript a un typage très dynamique, ce qui rend son utilisation plus accessible, ce qui peut, dans des cas plus complexes, poser des problèmes.

Malgré le faible typage de JavaScript, qui dans bien des cas, peut faciliter le codage, il est important pour nous de bien connaître les différentes structures de données natives, c'est-à-dire les types de données. Cela nous permettra de mieux comprendre comment fonctionne le traitement des données en JavaScript, et ainsi d'éviter des erreurs. Par ailleurs, cette compréhension vous sera indispensable dans votre parcours de développeur, car ce sont des notions que l'on va retrouver dans les autres langages de programmation.

Les types de données dits « *primitifs* » (qui ne comprennent pas les objets) définissent des valeurs que l'on ne peut pas changer (des valeurs immuables). Ces valeurs immuables s'appellent « *valeurs primitives* ».

Dans ce cours, nous nous intéresserons donc aux différents types de données en JavaScript, en nous focalisant sur les types primitifs (qui définissent les fameuses « *valeurs primitives* »). Nous ne parlerons donc pas des objets dans ce cours. Nous ferons appel à de nombreuses reprises à l'opérateur **typeof** permettant de renvoyer le type d'une donnée. Le cours sera composé de nombreux exemples de codes ainsi que de plusieurs exercices que vous pourrez réaliser vous-même en local ou via Replit.

II. Types numériques

A. Les types numériques

Le type Number

Le type **Number** permet de représenter des nombres. Cependant, les nombres représentables grâce au type **Number** sont compris dans un intervalle précis. Les nombres décimaux entre 2^{-1074} et 2^{1024} peuvent être représentés. L'intervalle comprenant les nombres entiers représentables d'une manière « *safe* » est plus restreint. Prenons une valeur numérique et affichons son type :

```
1 const nombre = 154.78;
2
3 console.log (typeof nombre);
```

On voit que la console affiche **number**. Donc **nombre** est bien de type **Number**.

On peut noter que les nombres entiers représentables de manière « *safe* » avec le type **Number** vont de $-(2^{53} - 1)$ à $2^{53} - 1$. Mais que se passe-t-il si on écrit un nombre entier très grand ? Voyons un exemple :

```
1 const nombre = 100071992549200345;
2
3 console.log (nombre);
```

On peut voir que la console affiche le nombre 100071992549200350. Mais ce n'est pas le même nombre. Pour quelle raison ? Sans rentrer dans les détails, ce problème d'arrondi est dû au fait que le nombre entier est trop grand pour être représenté par le type **Number**. Mais alors, comment représenter ce nombre ? En utilisant le type **Bigint**.

Le type BigInt

Le type **BigInt** permet de représenter des entiers plus grands que ceux représentables par le biais de **Number**. Essayons de représenter l'entier précédent avec le type **BigInt**. Pour cela, nous pouvons utiliser un **n** que nous ajoutons à la fin du nombre :

```
1 const nombre = 100071992549200345n;
2
3 console.log (nombre);
```

Là ça fonctionne, la console affiche : 100071992549200345n.

D'ailleurs, si on affiche le type de **nombre** :

```
1 const nombre = 100071992549200345n;
2
3 console.log (typeof nombre);
```

La console affiche : **bigint**. Le nombre est donc bien de type **BigInt**.

Quand on va réaliser des opérations arithmétiques, il faudra bien veiller à traiter ensemble des valeurs de même type, pour éviter les erreurs. Nous pouvons voir par exemple que si nous réalisons une addition d'un **BigInt** avec un **Number** :

```
1 const a = 16370197019808913673160913870317n;
2
3 const b = 16879361;
4
5 let addition = a + b;
6
7 console.log (addition);
```

Nous pouvons voir que ce code génère une erreur due à une incompatibilité de types dans une opération. On veillera donc à ce que les deux opérandes d'une opération mathématique soient du même type. Pour corriger ce code, nous pouvons par exemple faire de **b** un **BigInt** :

```
1 const a = 16370197019808913673160913870317n;
2
3 const b = 16879361n;
4
5 let addition = a + b;
6
7 console.log (addition);
```

À noter que la génération de **BigInt** doit être raisonnée, car avec de grands nombres, elle peut demander beaucoup de ressources et affecter les performances du programme.

NaN

NaN est un terme, une propriété qu'on va retrouver quand le résultat d'une opération arithmétique n'est pas un nombre. Ce terme signifie littéralement « *Not a Number* ». Si par exemple, nous essayons de multiplier un nombre par un texte :

```
1 const nombre = 12456 * "caractères";
2
3 console.log(nombre); //NaN
```

Un point intéressant en JS, c'est que si nous réalisons une opération arithmétique avec deux nombres dont un est une chaîne de caractères, JS convertit automatiquement la chaîne en nombre, et renvoie le résultat de l'opération :

```
1 const nombre = 12456 * "10";  
2  
3 console.log(nombre); //124560
```

Dans ce dernier cas, l'opération renverra 124560 et non **NaN**, puisque la chaîne « 10 » aura été convertie en nombre et que le calcul aura été opéré.

On peut noter que **NaN** est une valeur assez spéciale en JavaScript. En effet elle est unique, d'ailleurs, elle n'est même pas égale à elle-même :

```
1 console.log (NaN === NaN); //false
```

Voilà, vous connaissez maintenant les différentes valeurs primitives dédiées aux nombres en JavaScript.

B. Exercice

Question 1

[solution n°1 p.11]

Affichez le type de nombre dans la console.

```
1 const nombre = 45;  
2  
3 //code
```

Question 2

[solution n°2 p.11]

Définissez une variable nombre de type BigInt.

```
1 //code  
2  
3 console.log(typeof nombre);
```

Question 3

[solution n°3 p.11]

Corrigez le code pour qu'il n'y ait pas d'erreur.

```
1 const a = 678361083183098137091378n;  
2  
3 const b = 100003917097716398713970197;  
4  
5 console.log(a + b);
```

Question 4

[solution n°4 p.11]

Définissez nombre sur le nombre décimal 15.13.

```
1 const nombre;  
2  
3 console.log(nombre);
```

Question 5

[solution n°5 p.11]

Le code suivant affiche notre nombre arrondi. Corrigez le code pour que le nombre apparaisse tel quel.

```
1 const nombre =8473638490274657392736;  
2  
3 console.log (nombre);
```

III. Les autres types primitifs

A. Les autres types primitifs

Type Boolean

Vous connaissez déjà le type booléen. Une variable de type booléen peut avoir pour valeur **true** ou **false**. Donc pour définir une variable de type booléen, on peut simplement définir une variable sur **true** ou **false**. Par exemple :

```
1 const variable = true;
2
3 console.log(variable); //true
4
5 console.log(typeof variable); //boolean
```

Comme vous le savez, les booléens sont utilisés comme valeurs de retour de conditions. Ils permettent donc de comparer les données.

Type String

Le type **String** (chaîne de caractères) permet de représenter du texte. Les chaînes de caractères sont des ensembles de valeurs ordonnées, et indexées. La première valeur à l'indice 0, la seconde 1, etc. Pour définir une chaîne de caractères, nous pouvons définir une variable sur une valeur ou un ensemble de valeurs comprises entre guillemets :

```
1 const a = "caractères"
2
3 console.log(a); //caractères
4
5 console.log(typeof a); //string
```

Voyons un exemple de concaténation de chaînes avec l'opérateur **+=** par exemple :

```
1 let a = "caractères";
2
3 a += " écrit";
4
5 console.log(a); //caractères "écrit"
6
7 console.log(typeof a); //string
```

Ici ça fonctionne, on dirait qu'on modifie la chaîne "caractères". Mais vous vous en rappelez, nous avons dit que les types primitifs définissent des valeurs immuables, c'est-à-dire des valeurs qu'on ne peut pas modifier une fois définies. Pourtant, **String** est bien un type primitif qui définit les chaînes de caractères qui sont donc des valeurs primitives. "caractères" est une valeur primitive. Donc en théorie, on ne peut pas modifier la valeur d'une chaîne. Mais alors comment fonctionnent les opérateurs de concaténation ?

En réalité, la chaîne initiale n'est pas modifiée. L'opérateur **+=** va créer une nouvelle chaîne qui concatène les chaînes « caractères » et « écrit ». La variable **a** sera alors définie sur cette nouvelle chaîne, mais la chaîne de départ « caractères » n'aura pas été modifiée. C'est le même principe pour les opérateurs d'incrément ou d'affectation après addition avec les **numbers**.

Il est préconisé d'utiliser les chaînes de caractères uniquement pour les textes.

Type Symbol

Un symbole est une valeur unique et qui ne peut pas être changée. Les symboles peuvent être utilisés pour créer des clés uniques ciblant une propriété d'un objet, et ainsi éviter les conflits pouvant exister avec des propriétés créées par d'autres instructions de code. C'est peut-être un peu complexe à comprendre comme ça, mais concentrons-nous sur comment créer un symbole. Pour créer un symbole, il nous faut utiliser une fonction : **Symbol()**. Cette fonction va nous permettre de créer une valeur de type **Symbol**.

```
1 let sy1 = Symbol('marque');
2
3 console.log(sy1); //Symbol(marque)
4
5 console.log(typeof sy1); //symbol
```

On voit qu'on crée un symbole **sy1** défini sur **Symbol('marque')**. Créons un deuxième symbole que nous définirons aussi sur **Symbol('marque')**, et comparons ces valeurs :

```
1 let sy1 = Symbol('marque');
2
3 let sy2 = Symbol('marque');
4
5 console.log (sy1 === sy2); //false
```

C'est intéressant, on définit les deux variables sur la même expression, et les deux symboles ne sont pas égaux. Ça confirme bien qu'un symbole a la caractéristique d'être unique.

Les symboles vont nous permettre de créer des clés de propriétés uniques et non énumérables (ces propriétés ne seront pas répertoriées lorsqu'on parcourra les propriétés d'un objet avec la boucle **for/in**). C'est un concept peut être un peu complexe à comprendre pour l'instant, mais ne vous inquiétez pas, contentez-vous pour l'instant de retenir que les symboles nous serviront dans la création de clés de propriétés.

Type Null

La valeur **null** est une valeur primitive représentant simplement l'absence de valeurs. **null** doit être définie pour indiquer l'absence de valeur. Par exemple, si nous écrivons :

```
1 let variable = null;
2
3 console.log (variable); //null
```

La console affiche **null**. On peut noter un bug si on cherche à afficher le type de **variable** :

```
1 let variable = null;
2
3 console.log (typeof variable); //object
```

On peut voir que le type renvoyé n'est pas **null** mais **object**. C'est un bug de JavaScript qui existe depuis pas mal de temps, mais qui n'a pas été corrigé visiblement pour des raisons de compatibilité.

Type Undefined

À la différence de **null**, la valeur primitive **undefined** (définie par le type **undefined**) est renvoyée par défaut lorsqu'on cherche à récupérer une valeur non définie ou une propriété inexistante. Par exemple, si nous écrivons :

```
1 let variable;
2
3 console.log (variable); //undefined
4
5 console.log (typeof variable); //undefined
```

Nous pouvons voir que c'est automatiquement ce qui est renvoyé par l'expression **variable**, cette variable n'étant pas définie. À la différence du type **null**, nous n'avons pas besoin de définir ma variable sur **undefined** pour qu'elle renvoie **undefined**, c'est l'expression renvoyée par défaut par JavaScript.

Voilà, vous connaissez maintenant les types primitifs en JavaScript.

B. Exercice

Question 1

[solution n°6 p.12]

Définir une constante **a** sur un booléen :

```
1 //
2
3 console.log(typeof a);
```

Question 2

[solution n°7 p.12]

Définir une constante **a** sur une chaîne de caractères :

```
1 //code
2
3 console.log(typeof a);
```

Question 3

[solution n°8 p.12]

Modifier le code pour que la console affiche **null** :

```
1 let a;
2
3 console.log(a);
```

Question 4

[solution n°9 p.12]

Créer un symbole :

```
1 sy1 = /*code*/;
2
3 console.log (typeof sy1);
```

Question 5

[solution n°10 p.12]

Concaténez « *Martin* » et « Dupont » afin d'obtenir « *Martin Dupont* » et indiquez le type de name.

```
1 let name= "Martin";
2
3 /*code*/;
4
5 console.log(/*code*/);
6
7 console.log(/*code*/);
```

IV. Essentiel

Dans ce cours, nous nous sommes intéressés aux différents types de données en JavaScript, en nous concentrant sur les types primitifs. Les types primitifs sont les types de données qui définissent des valeurs primitives, donc des valeurs immuables. Les objets ne sont donc pas des types primitifs.

Nous avons vu que JavaScript fournit 2 types de données dédiées aux nombres : le type **Number** et le type **BigInt**. Le type **BigInt** permet de stocker des nombres entiers très grands. Le type **Number** quant à lui permet de stocker des nombres de différents types (entiers et décimaux, etc.).

Nous nous sommes également penchés sur d'autres types primitifs que vous connaissiez pour certains déjà : le type **Boolean** permettant de stocker **true** ou **false**, le type **String** permettant de définir des chaînes de caractères (à utiliser pour les textes), le type **Symbol** permettant notamment de créer des clés vers des propriétés, le type **Null** qui permet d'indiquer l'absence de valeur et le type **Undefined** qui renvoie la valeur **undefined** par défaut lorsqu'on cherche à récupérer une valeur non définie.

Par ailleurs, vous savez désormais que le type **Null** doit être défini, à la différence de **Undefined** dont la valeur primitive est renvoyée par défaut, dans le cas où une valeur n'est pas définie. Un bug de JavaScript est visible lorsqu'on affiche la valeur renvoyée par **typeof null**, on peut voir que le type renvoyé est **object** et non **null**. Mais une fois qu'on le sait, ce n'est pas très handicapant.

V. Auto-évaluation

A. Exercice

Vous cherchez à réaliser un script permettant, à l'aide d'un **switch**, de vérifier le type d'une valeur (pour toutes les valeurs primitives abordées dans ce cours). Si le type est **String**, la console affichera : "C'est une chaîne de caractères", si c'est un **Number**, "C'est un nombre", si c'est un **Bigint**, "C'est un grand entier", etc. Les types à référencer sont :

number

bigint

boolean

string

symbol

undefined

null

Voici votre code de base :

```
1 let variable;  
2  
3 //code;
```

Question 1

[solution n°11 p.13]

Réaliser le script en question, sans référencer le type **null**.

Question 2

[solution n°12 p.13]

Nous n'avons pas référencé le cas où **variable** serait définie sur **null** car, un bug en JavaScript fait que l'expression **typeof null** renvoie **object** et non **null**. Trouver une combine pour que dans le cas où **variable** est **null**, la console affiche "C'est une valeur nulle".

B. Test

Exercice 1 : Quiz

[solution n°13 p.14]

Question 1

Parmi les exemples suivants, quelle est la syntaxe correcte permettant de définir un **BigInt** ?

- ☐ let var = 1269812091309137891630813691n;
- ☐ let var = 1269812091309137891630813691;
- ☐ let var = bigint 1269812091309137891630813691n;

Question 2

Parmi les exemples suivants, quelle est la syntaxe correcte permettant de définir un booléen ?

- ☐ let var = "true";
- ☐ let var = true;
- ☐ let var = boolean "true"

Question 3

Parmi les exemples suivants, quelle est la syntaxe correcte permettant de définir un symbole ?

- ☐ let var = Symbol ("nom");
- ☐ let var = "nom";
- ☐ let var = Symbol nom;

Question 4

Voici un exemple de code :

```
1 let a = 15;  
2  
3 a += 5;  
4  
5 console.log (a); //20
```

La valeur primitive 15 :

- ☐ Est modifiée
- ☐ N'est pas modifiée
- ☐ Est décrémentée

Question 5

Parmi les exemples suivants, lequel est correct pour définir une chaîne de caractères ?

- ☐ let var = "caractères";
- ☐ let var = (caractères);
- ☐ let var = caractères;

Solutions des exercices

p. 5 Solution n°1**Sortie attendue :** number**Solution :**

```
1 const nombre = 45;  
2  
3 console.log(typeof nombre);
```

p. 5 Solution n°2**Sortie attendue :** bigint**Solution possible :**

```
1 const nombre = 678361083183098137091378n;  
2  
3 console.log(typeof nombre);
```

p. 5 Solution n°3**Sortie attendue :** 100682278180899496851061575**Solution :**

```
1 const a = 678361083183098137091378n;  
2  
3 const b = 100003917097716398713970197n;  
4  
5 console.log(a + b);
```

p. 5 Solution n°4**Sortie attendue :** 15.13**Solution :**

```
1 const nombre = 15.13;  
2  
3 console.log(nombre);
```

p. 5 Solution n°5**Sortie attendue :** 8473638490274657392736**Solution :**

```
1 const nombre = 8473638490274657392736n;  
2  
3 console.log (nombre);
```

p. 8 Solution n°6

Sortie attendue : boolean

Solution possible :

```
1 const a = true;
2
3 console.log(typeof a) ;
```

p. 8 Solution n°7

Sortie attendue : string

Solution possible :

```
1 const a = "caractères";
2
3 console.log(typeof a) ;
```

p. 8 Solution n°8

Sortie attendue : null

Solution :

```
1 let a = null;
2
3 console.log(a);
```

p. 8 Solution n°9

Sortie attendue : symbol

Solution possible :

```
1 sy1 = Symbol("nom");
2
3 console.log(typeof sy1);
```

p. 8 Solution n°10

Sortie attendue :

Martin Dupont

string

Solution :

```
1 let name= "Martin";
2
3 name += " Dupont";
4
5 console.log(name);
6
7 console.log(typeof name);
```

p. 9 Solution n°11

Voici un code qui fonctionne :

```
1 let variable;
2
3 switch (typeof variable) {
4   case "number":
5     console.log("C'est un nombre");
6     break;
7   case "bigint":
8     console.log("C'est un grand entier");
9     break;
10  case "boolean":
11    console.log("C'est un booléen");
12    break;
13  case "string":
14    console.log("C'est une chaîne de caractères");
15    break;
16  case "symbol":
17    console.log("C'est un symbole");
18    break;
19  case "undefined":
20    console.log("C'est une valeur non définie");
21    break;
22  default:
23    console.log("Ce type n'est pas référencé");
24    break;
25 }
```

p. 9 Solution n°12

On peut modifier la clause **default** pour insérer une condition **if(variable == null)** :

```
1 let variable;
2
3 switch (typeof variable) {
4   case "number":
5     console.log("C'est un nombre");
6     break;
7   case "bigint":
8     console.log("C'est un grand entier");
9     break;
10  case "NaN":
11    console.log("Ce n'est pas un nombre");
12    break;
13  case "boolean":
14    console.log("C'est un booléen");
15    break;
16  case "string":
17    console.log("C'est une chaîne de caractères");
18    break;
19  case "symbol":
20    console.log("C'est un symbole");
21    break;
22  case "undefined":
23    console.log("C'est une valeur non définie");
24    break;
```

```


25 default:
26   if(variable == null) {
27     console.log("La valeur est nulle");
28   }
29   else {
30     console.log("Ce type n'est pas référencé");
31   }
32   break;
33 }

```

Exercice p. 9 Solution n°13


Question 1

Parmi les exemples suivants, quelle est la syntaxe correcte permettant de définir un **BigInt** ?

- ☒ let var = 1269812091309137891630813691n;
- ☐ let var = 1269812091309137891630813691;
- ☐ let var = bigint 1269812091309137891630813691n;
-  Pour définir un **BigInt**, on peut définir la variable sur le grand entier avec un « **n** » à la fin.


Question 2

Parmi les exemples suivants, quelle est la syntaxe correcte permettant de définir un booléen ?

- ☐ let var = "true";
- ☒ let var = true;
- ☐ let var = boolean "true"
-  Pour définir un booléen, il faut simplement définir une variable sur **true** ou **false** (sans guillemets, car ce n'est pas une chaîne de caractères).

Question 3

Parmi les exemples suivants, quelle est la syntaxe correcte permettant de définir un symbole ?

- ☒ let var = Symbol ("nom");
- ☐ let var = "nom";
- ☐ let var = Symbol nom;
-  Pour définir un symbole, on peut utiliser la fonction **Symbol**.

Question 4

Voici un exemple de code :


```

1 let a = 15;
2
3 a += 5;
4
5 console.log (a); //20

```


La valeur primitive 15 :

- ☐ Est modifiée
- ☒ N'est pas modifiée
- ☐ Est décrémentée

 En JavaScript, les types primitifs définissent des valeurs primitives immuables, à la différence des objets. On ne peut donc pas modifier une valeur primitive. En réalité, 15 qui est de type **number** n'est donc pas modifiée. L'opérateur d'affectation après addition crée une nouvelle valeur primitive de type **number** qui stockera le résultat de 15 + 5, donc 20. La variable **a** est alors définie sur la valeur de cette nouvelle valeur primitive.

Question 5

Parmi les exemples suivants, lequel est correct pour définir une chaîne de caractères ?

- ☒ `let var = "caractères";`
 - ☐ `let var = (caractères);`
 - ☐ `let var = caractères;`
-  Pour définir une chaîne de caractères, il faut simplement définir une variable sur un ensemble de caractères entre guillemets.