

L'objet JavaScript Number

Table des matières

| | |
|--|-----------|
| I. Contexte | 3 |
| II. Découverte de l'objet JS Number | 3 |
| A. Découverte de l'objet JS Number | 3 |
| B. Exercice | 6 |
| III. Méthodes de Number | 7 |
| A. Méthodes de Number | 7 |
| B. Exercice | 10 |
| IV. Essentiel | 10 |
| V. Auto-évaluation | 11 |
| A. Exercice | 11 |
| B. Test | 11 |
| Solutions des exercices | 12 |

I. Contexte

Durée : 1 h

Prérequis : avoir étudié le cours jusqu'ici

Environnement de travail : Replit

Contexte

Les types primitifs sont des types qui définissent des valeurs primitives immuables. En JavaScript, il existe deux types primitifs permettant de définir des nombres : **Number** et **BigInt**. Le type **Number** permet de représenter des nombres entiers et décimaux dans une certaine limite. Le type **BigInt** est quant à lui dédié aux grands entiers.

Vous savez par ailleurs que JavaScript permet de développer en Programmation Orientée Objet. Or, est-il possible de manipuler des nombres comme des objets ? Et avant tout, qu'est-ce qu'un objet ?

Dans ce cours, nous parlerons de l'objet JavaScript **Number**, à ne pas confondre avec le type primitif **Number**. Nous verrons à quoi il correspond et aborderons plusieurs de ses caractéristiques essentielles. Nous parlerons brièvement du concept d'objet afin de mieux visualiser ce qu'il représente.

Ce cours sera constitué de nombreux exemples de codes, ainsi que d'exercices pratiques réalisables via l'interface Replit. N'hésitez donc pas à tester les exemples de code et à réaliser les exercices, ils vous permettront de plus rapidement comprendre les concepts traités.

II. Découverte de l'objet JS Number

A. Découverte de l'objet JS Number

Avant de commencer à parler spécifiquement de l'objet JS **Number**, faisons le point sur la notion d'objet en Programmation Orientée Objet. C'est une notion essentielle en programmation.

Explication du concept d'objet

Un objet est une entité qui se caractérise par un type qui le définit, ainsi que des propriétés. C'est exactement le même principe que dans la réalité. Prenons un objet exemple : un ordinateur, plus précisément un ordinateur de la marque HP. Voici les caractéristiques de notre objet :

Type : Ordinateur

Propriétés :

- Marque = « HP »
- Ram = 8
- Stockage = 256
- Processeur = « Intel »
- Démarrage : démarrage de l'ordinateur
- Extinction : extinction de l'ordinateur

Nous pouvons constater que notre objet est de type Ordinateur. Il a plusieurs propriétés (appelées aussi attributs) qui correspondent aux différentes informations caractérisant notre objet. Mais nous pouvons voir que les deux dernières propriétés ne sont pas de simples valeurs, ce sont des mécanismes, déclenchables via l'objet (il est possible de démarrer et d'éteindre l'ordinateur).

De manière très similaire, nous pouvons dire que les objets ont des propriétés qui peuvent être soit :

- Des variables, définies sur une valeur (appelées souvent attributs en POO).
- Des fonctions, c'est-à-dire des ensembles d'instructions qu'il est possible d'appeler via notre objet. Ces fonctions (associées à des objets) s'appellent des **méthodes**.

Exemple

Si nous traduisons en code le descriptif de notre ordinateur, nous pouvons l'écrire ainsi :

```
1 let ordi1 = new Object(); //nous construisons notre objet
2
3 ordi1 = {
4   marque: "HP",
5
6   ram: 8,
7
8   stockage: 256,
9
10  processeur: "Intel",
11
12  demarrage: function() {
13    console.log("Démarrage de l'ordinateur");
14  },
15
16  extinction: function() {
17    console.log("Extinction de l'ordinateur");
18  }
19 }
```

Ici, nous pouvons voir que nous créons un objet que nous stockons via la variable **ordi1**. Ne vous attardez pas pour l'instant sur l'expression **new Object()**, nous aborderons ce point en temps voulu. Comprenez simplement que cette instruction permet de créer un objet. Puis, nous définissons les propriétés de notre objet parmi lesquelles figurent les attributs (les variables), ainsi que les méthodes **demarrage()** et **extinction()**. Nous pouvons voir que pour définir les propriétés, nous utilisons une syntaxe spécifique. Il existe toutefois d'autres moyens de procéder, mais nous ne les verrons pas dans ce cours pour ne pas nous égarer.

Nous pouvons donc accéder aux différentes propriétés de notre objet **ordi1**, simplement en écrivant **ordi1** suivi d'un point « . » et du nom de la propriété.

Exemple

Prenons un exemple :

```
1 let ordi1 = new Object(); //nous construisons notre objet
2
3 ordi1 = {
4   marque: "HP",
5
6   ram: 8,
7
8   stockage: 256,
9
10  processeur: "Intel",
11
12  demarrage: function() {
13    console.log("Démarrage de l'ordinateur");
14  },
15 }
```

```
16 extinction: function() {  
17     console.log("Extinction de l'ordinateur");  
18 }  
19 }  
20  
21 console.log(ordi1.marque); //HP  
22  
23 console.log(ordi1.ram); //8  
24  
25 ordi1.demarrage(); //Démarrage de l'ordinateur
```

Remarque

Si nous affichons le type de notre objet avec : **console.log(typeof ordi1)**, la console va afficher **object** et non « *Ordinateur* ». C'est tout à fait normal, nous avons construit un objet via l'expression **new Object()**, nous faisons appel au constructeur **Object**.

Nous verrons rapidement ce qu'est un constructeur dans les paragraphes suivants. Simplement, dans notre exemple, nous n'avons pas défini de sous-type **Ordinateur**, mais nous avons créé un objet de type **Object**, ce qui explique le résultat de : **console.log(typeof ordi1)**.

Les objets vont donc permettre de manipuler les données d'une manière différente, présentant de nombreux avantages que vous découvrirez au fur et à mesure de votre parcours JavaScript.

Introduction à l'objet Number

Maintenant que nous avons compris globalement ce qu'est un objet, intéressons-nous à l'objet **Number**. Comme vous le savez, le type primitif **Number** définit des valeurs primitives immuables. Cependant, par défaut, un objet n'est pas immuable, car il est possible de modifier ses propriétés après l'avoir créé. Mais alors qu'est-ce que l'objet **Number** ?

Un objet **Number** est un objet qui enveloppe un type primitif **Number**. C'est donc un objet constitué de propriétés, c'est un objet muable, mais qui stocke un type primitif **Number**. Un objet **Number** permet donc de manipuler des nombres comme des objets. Avant de voir quelques caractéristiques de l'objet **Number**, voyons comment créer un objet **Number**.

Méthode Créer un objet Number

Pour créer un objet **Number**, il va nous falloir appeler une méthode dont nous avons parlé très brièvement : un constructeur. Les constructeurs sont des méthodes permettant comme leur nom l'indique de construire un objet.

Exemple

Pour appeler le constructeur de **Number**, nous pouvons procéder ainsi :

```
1 const nombre = new Number(1426.156)  
2 console.log(nombre);
```

Dans cet exemple, nous créons un objet **Number**, en appelant le constructeur de **Number**. Le mot clé **new** est indispensable. Nous passons comme argument du constructeur le nombre qui sera enveloppé dans notre objet.

Les valeurs possibles de Number

Un objet **Number** peut représenter des valeurs numériques comprises dans certaines limites. Ces limites sont définies dans des propriétés par défaut de **Number**. Pour les récupérer, nous pouvons les afficher via le constructeur **Number** :

```
1 console.log(Number.MIN_VALUE);
2
3 console.log(Number.MAX_VALUE);
```

Number.MIN_VALUE est la valeur minimale qu'un objet **Number** peut contenir. **Number.MAX_VALUE** est à l'inverse la valeur maximale qu'un objet **Number** peut contenir.

Méthode Opérations arithmétiques

Nous pouvons réaliser sans problème des opérations arithmétiques avec des objets **Number**. Pourtant, les objets ne sont pas des valeurs primitives numériques. Nous ne pouvons pas normalement réaliser des opérations arithmétiques entre deux objets, mais uniquement entre deux valeurs numériques. Nous verrons en fin de seconde partie comment JavaScript rend possibles les opérations arithmétiques avec comme opérandes des objets **Number**, comme dans cet exemple :

Exemple

```
1 const a = new Number(18);
2
3 const b = new Number(17);
4
5 console.log(a + b); //35
6
7 console.log (a - b); //1
8
9 console.log(a * b); //306
10
11 console.log (a % b); //1
```

Il est donc possible de réaliser des opérations arithmétiques très facilement avec les objets **Number**.

B. Exercice

Question 1

[solution n°1 p.13]

Créer un objet **Number** dont la valeur est de 1256753 :

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

Question 2

[solution n°2 p.13]

Créer une fonction permettant de renvoyer le produit de deux objets **Number**.

```
1 const a = new Number(10);
2
3 const b = new Number(12);
4
5 //function
6
7 console.log(/*appel*/) ;
```

Question 3

[solution n°3 p.13]

Afficher dans la console la valeur maximale représentable via un objet **Number**.

```
1 console.log(/*code*/) ;
```

Question 4

[solution n°4 p.13]

Afficher dans la console la valeur minimale représentable via un objet **Number**.

```
1 console.log(/*code*/) ;
```

Question 5

[solution n°5 p.13]

Créer un objet **Number** à partir de la chaîne de caractères **nombre**. L'objet **Number** sera stocké via la variable **nb**.

```
1 const nombre = "5675";
2
3 const nb = /*code*/
4
5 console.log(nb);
```

III. Méthodes de Number**A. Méthodes de Number**

Maintenant que nous comprenons un peu mieux le concept d'objet **Number**, parlons de quelques-unes de ses méthodes.

Nous pouvons distinguer deux types de méthodes : les méthodes statiques, c'est-à-dire les méthodes que nous pouvons appeler directement via le mot **Number**, et les méthodes accessibles via les objets créés grâce au constructeur **Number**. Les objets créés grâce à un constructeur sont appelés des « instances ». Donc, pour faire simple, il existe :

- Les méthodes dites « *statiques* », que nous pouvons appeler directement via le mot **Number**.
- Les méthodes non statiques, que nous pouvons appeler via un objet, c'est-à-dire est une instance de **Number**.

Les méthodes statiques n'ont donc pas besoin d'instance de **Number** pour être appelées. Voyons déjà quelques méthodes statiques.

Méthode **Les méthodes Number.is...()**

Intéressons-nous à 4 méthodes statiques :

- **Number.isNaN()** [**developer.mozilla.org : Number isNaN\(\)**](https://developer.mozilla.org : Number isNaN())¹
- **Number.isFinite()** [**developer.mozilla.org : Number/isFinite\(\)**](https://developer.mozilla.org : Number/isFinite())²
- **Number.isInteger()** [**developer.mozilla.org : Number/isInteger\(\)**](https://developer.mozilla.org : Number/isInteger())³
- **Number.isSafeInteger()** [**developer.mozilla.org : Number/isSafeInteger\(\)**](https://developer.mozilla.org : Number/isSafeInteger())⁴

Chacune de ses méthodes va renvoyer un booléen. La première va renvoyer **true** si l'expression passée comme argument n'est pas un nombre. La seconde va renvoyer **true** si le nombre passé en argument est fini. La troisième va renvoyer **true** si le nombre est un entier et la quatrième si le nombre est un entier représentable de manière « *safe* » avec **Number**.

1 https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN

2 https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Number/isFinite

3 https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Number/isInteger

4 https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Number/isSafeInteger

Exemple

Voici quelques exemples d'appel de ces méthodes statiques :

```
1 console.log(Number.isNaN(62378)); //false
2
3 console.log(Number.isFinite(5817629)); //true
4
5 console.log(Number.isInteger(5817629.872)); //false
6
7 console.log(Number.isSafeInteger(5817629)); //true
```

L'objet Number.prototype

Parlons maintenant de méthodes accessibles via des objets créés via le constructeur **Number**, donc des instances de **Number**. Ces méthodes ne sont donc pas statiques. Pour bien comprendre, il nous faut nous intéresser rapidement au concept de **prototype**.

En JavaScript, chaque objet a une propriété appelée **prototype**. Cette propriété est en fait un objet qui contient les propriétés héritées par chaque instance, par chaque objet de son type. Le **prototype** est en quelques sortes un modèle pour les instances, c'est-à-dire les objets d'un type. Donc, la propriété **Number.prototype** est un objet qui contient des propriétés, dont des méthodes accessibles à toutes les instances de **Number**, ainsi qu'à tous les objets **Number**.

Il est possible d'afficher les propriétés principales de l'objet **Number.prototype** en utilisant la méthode **Object.getOwnPropertyNames()** :

```
1 console.log(Object.getOwnPropertyNames(Number.prototype));
```

Nous pouvons constater que la console affiche différentes propriétés (qui sont en l'occurrence des méthodes) de l'objet **Number.prototype**.

Ces méthodes sont donc héritées par toutes les instances de **Number**. Nous pouvons donc accéder aux méthodes héritées du prototype en écrivant : **nomDeLobjet.nomDeLaMethode()**, JavaScript fait le reste. Voyons quelques exemples de méthodes.

Méthode Les méthodes to...()

Voyons deux exemples de méthodes :

- **toExponential()**
- **toString()**

La méthode **toExponential()** permet de renvoyer une chaîne contenant le nombre stocké dans l'objet **Number** avec une notation exponentielle (avec une puissance). Nous pouvons passer en argument le nombre de chiffres décimaux (chiffres après la virgule). Par défaut, si nous ne passons pas d'arguments, la méthode considérera le nombre de chiffres décimaux nécessaires pour que le nombre soit représenté. Si nous spécifions un argument, nous pouvons nous attendre à ce que le résultat soit un nombre arrondi.

Exemple

Par exemple :

```
1 const nombre = new Number(13578);
2
3 console.log(nombre.toExponential()); //1.3578e+4
4
5 console.log(nombre.toExponential(2)); //1.36e+4
```


Dans le premier appel de la méthode, aucun argument n'est passé. La méthode utilise le nombre de chiffres décimaux nécessaires pour représenter le nombre avec la notation exponentielle (ici 4 : 3578). Dans le second appel, nous passons comme argument 2. Donc, le résultat contient deux chiffres décimaux : 36. Nous pouvons voir qu'il y a un arrondi, le résultat est donc moins précis.

La méthode **toString()** permet quant-à-elle de renvoyer une chaîne de caractères contenant la valeur du nombre de l'objet **Number**.

Exemple

Prenons un exemple :

```
1 const nombre = new Number(13578);
2
3 let nombreChaine = nombre.toString();
4
5 console.log(nombreChaine); //13578
6
7 console.log(typeof nombreChaine); //string
```

Nous pouvons voir que le type de **nombreChaine** est bien **string**, c'est une chaîne de caractères.

Méthode La méthode ValueOf

Dans le dernier exemple, nous avons récupéré la valeur du nombre d'un objet **Number** en le convertissant en chaîne de caractères. Mais dans certains cas, nous voudrions récupérer la valeur de notre nombre avec le type primitif **Number**, et non sous la forme d'une chaîne de caractères.

Exemple

Prenons un exemple. Si nous faisons :

```
1 const nombre = new Number(13578);
2
3 let nombreRecup = nombre;
4
5 console.log(nombreRecup); //Number 13578
```

Nous voyons que **nombreRecup** est une référence vers l'objet, et non vers la valeur de son nombre. **nombreRecup** est donc un objet, et non une valeur primitive de type **Number**. En fait, c'est la méthode **valueOf()** du prototype qui va nous permettre de récupérer la valeur du nombre, avec le type primitif **Number**. Prenons un exemple :

```
1 const nombre = new Number(13578);
2
3 let nombreRecup = nombre.valueOf();
4
5 console.log(nombreRecup); //13578
6
7 console.log(typeof nombreRecup); //number
```

Donc, avec la méthode **valueOf()**, nous pouvons récupérer la valeur primitive qui a été enveloppée dans l'objet **Number**. La méthode **valueOf()** va être essentielle à utiliser dans les cas où nous voulons accéder au nombre stocké dans un objet **Number**.

Voilà, nous venons de voir quelques méthodes accessibles via chaque instance de **Number**.

Complément **Fonctionnement des opérations arithmétiques avec des objets Number**

B. Exercice

Question 1

[solution n°6 p.13]

Créer une fonction permettant de savoir si un objet **Number** est un entier, et afficher sa valeur de retour dans la console avec l'objet **nombre** passé comme argument. La valeur de retour de la fonction sera un booléen.

```
1 const nombre = new Number(1678);
2
3 //code
```

Question 2

[solution n°7 p.14]

Créer une fonction permettant de retourner un nombre avec écriture exponentielle (avec 4 chiffres décimaux max). Afficher sa valeur de retour en passant **nombre** en argument.

```
1 const nombre = new Number(3167830918);
2
3 //code
```

Question 3

[solution n°8 p.14]

Nous voulons afficher dans la console le nombre de caractères (chiffres) d'un nombre. Pour cela, nous voulons convertir sa valeur en chaîne de caractères pour afficher la propriété **length** qui correspond au nombre de caractères d'une chaîne. Corriger le code pour qu'il fonctionne :

```
1 const nombre = new Number(3167830918);
2
3 console.log(Number.toString(nombre).length)
```

Question 4

[solution n°9 p.14]

Nous voulons afficher dans la console un booléen indiquant si **nombre** contient un nombre fini (avec un nombre fini de chiffres). Ce code renvoie **false** alors que le nombre stocké par l'objet **nombre** est fini. Corriger le code pour qu'il fonctionne :

```
1 const nombre = new Number(8909.61892);
2
3 console.log(Number.isFinite(nombre));
```

Question 5

[solution n°10 p.14]

Créer une fonction permettant de déterminer si un nombre est un multiple de 10 ou non. Cette fonction permettra de passer un nombre en paramètre qui sera un objet **Number**. La valeur de retour sera un booléen. Une fois la fonction définie, l'appeler en passant **nombreTest** en argument.

```
1 const nombreTest = new Number(15267);
2
3 //définition de la fonction
4
5 console.log(/*valeur de retour de la fonction*/);
```

IV. Essentiel

Dans ce cours, nous avons parlé de l'objet JS **Number**. Les objets sont des entités permettant de manipuler les données d'une manière spécifique. Chaque objet se caractérise par un ensemble de propriétés, qui seront de simples variables (des attributs), ou des méthodes (qui sont finalement des fonctions).

L'objet **Number** est un objet qui permet d'envelopper un type primitif **Number** dans un objet. Il permet de traiter et de manipuler les nombres comme des objets. Nous avons pu voir que les objets **Number** peuvent stocker des nombres compris dans une certaine limite.

Pour construire un objet **Number**, il faut utiliser le constructeur **Number**, en le précédant du mot clé **new**, et en passant comme argument la valeur primitive (le nombre) stockée. Nous avons vu qu'il est très facile de réaliser des opérations arithmétiques avec des objets de type **Number**.

Number donne accès à des méthodes statiques, c'est-à-dire des méthodes qu'on peut appeler directement via le mot **Number**. Les méthodes commençant par « *is* » permettent de renvoyer un booléen pour déterminer si un nombre est entier, fini, etc.

Chaque instance de **Number**, c'est-à-dire chaque objet créé avec le constructeur **Number** a accès à des méthodes (non statiques) héritées de l'objet **Number.prototype**. Nous avons abordé dans ce cours la méthode plus qu'essentielle **valueOf()** permettant de récupérer la valeur du type primitif (du nombre) stocké par l'objet **Number**. Nous avons par ailleurs parlé des méthodes de conversion **toString()** et **toExponential()**.

Voilà, le concept d'objet en POO vous est moins étranger, et vous connaissez maintenant l'objet **Number**. Il vous sera très utile lorsque vous traiterez des données numériques en JavaScript.

V. Auto-évaluation

A. Exercice

Vous êtes gérant d'un magasin d'informatique et vous missionnez des employés de générer des étiquettes de prix pour plusieurs produits. Vous souhaitez créer un programme permettant aux employés de savoir si un prix est valide ou non.

Vos articles doivent avoir un prix entier, et être compris entre 500 (inclus) et 1 500 € (inclus).

Question 1

[solution n°11 p.15]

Réaliser un code avec une fonction permettant d'indiquer si un prix stocké dans un objet **Number** est valide.

Question 2

[solution n°12 p.15]

Insérer dans la condition un critère vérifiant que l'argument passé est bien un objet et non un type primitif.

B. Test

Exercice 1 : Quiz

[solution n°13 p.15]

Question 1

Quelle méthode permet de vérifier si un objet **Number** stocke un nombre entier ?

- ☐ `isInteger()`
- ☐ `isNaN()`
- ☐ `toFixed()`

Question 2

Quelle propriété contient la valeur maximale qu'un objet **Number** peut stocker ?

- ☐ `Number.MAX_VALUE`
- ☐ `Number.MIN_VALUE`
- ☐ `Number.isFinite()`

Question 3

Quelle méthode permet de convertir un objet **Number** en une chaîne de caractères ?

- ☐ toString()
- ☐ valueOf()
- ☐ toPrecision()

Question 4

Quelle est la méthode qui permet de récupérer l'écriture exponentielle d'un nombre ?

- ☐ toString()
- ☐ toExponential()
- ☐ toFixed()
- ☐ toPrecision()

Question 5

Quelle est la méthode statique qui permet de vérifier uniquement si un nombre est fini ou non ?

- ☐ isFinite()
- ☐ isNaN()
- ☐ isInteger()

Solutions des exercices

p. 6 Solution n°1**Sortie attendue :** Number 1256753

```
1 let nombre = new Number(1256753);
2
3 console.log(nombre);
```

p. 6 Solution n°2**Sortie attendue :** 120

```
1 const nb = new Number(10);
2
3 const nb2 = new Number(12);
4
5 function mul(a, b) {
6   return a * b;
7 }
8
9 console.log(mul(nb, nb2));
```

p. 7 Solution n°3**Sortie attendue :** 1.7976931348623157e+308

```
1 console.log(Number.MAX_VALUE);
```

p. 7 Solution n°4**Sortie attendue :** 5e-324

```
1 console.log(Number.MIN_VALUE);
```

p. 7 Solution n°5**Sortie attendue :** Number 5675

```
1 const nombre = "5675";
2
3 const nb = new Number(nombre);
4
5 console.log(nb);
```

p. 10 Solution n°6

Sortie attendue : true

```
1 const nombre = new Number(1678);
2
3 function entier (nb) {
4   return Number.isInteger(nb.valueOf());
5 }
6
7 console.log(entier(nombre));
```

La fonction renverra la valeur de retour de la méthode statique **Number.isInteger()** en passant comme argument la valeur de retour de **nb.valueOf()**. Nous utilisons la méthode **valueOf()** pour bien récupérer le nombre stocké dans l'objet **Number**, car c'est lui qui est important, et non l'objet lui-même. Si nous avons passé **nb** comme argument de **Number.isInteger()**, la fonction n'aurait pas fonctionné, car c'est un nombre qui doit être passé comme argument de cette méthode, et non un objet.

p. 10 Solution n°7

Sortie attendue : 3.1678e+9

```
1 const nombre = new Number(3167830918);
2
3 function exp(nb) {
4   return nb.toExponential(4);
5 }
6
7 console.log(exp(nombre));
```

p. 10 Solution n°8

Sortie attendue : 10

```
1 const nombre = new Number(3167830918);
2
3 console.log(nombre.toString().length)
```

La méthode **toString()** n'est pas statique, elle est héritée de l'objet **Number.prototype** et est donc accessible via les instances de **Number**, et non directement via **Number**. Nous appelons cette méthode via l'objet **nombre** qui est une instance de **Number**, puis, nous affichons la valeur de retour de la propriété **length** de notre chaîne.

p. 10 Solution n°9

Sortie attendue : true

```
1 const nombre = new Number(8909.61892);
2
3 console.log(Number.isFinite(nombre.valueOf()));
```

Il nous faut utiliser la méthode **valueOf()** afin de récupérer la valeur du nombre stocké, et non l'objet **Number** directement.

p. 10 Solution n°10

Sortie attendue : false

```
1 const nombreTest = new Number(15267);
2
3 function isMultipleDix(nb) {
4   return (nb.valueOf() % 10 == 0);
5 }
6
7 console.log(isMultipleDix(nombreTest));
```

Pareillement, nous utilisons la méthode **valueOf()** pour récupérer la valeur du nombre stocké par l'objet **Number** passé comme argument.

p. 11 Solution n°11

Voici un code qui fonctionne :

```
1 const prix = new Number(500);
2
3 function verifPrice(pr) {
4   if(Number.isInteger(pr.valueOf()) && pr.valueOf() >= 500 && pr.valueOf() <= 1500) {
5     console.log("Le prix est valide");
6   }
7   else {
8     console.log("Le prix n'est pas valide");
9   }
10 }
11
12 verifPrice(prix);
```

Nous pouvons voir qu'à chaque fois qu'on cherche à récupérer le prix, nous utilisons la méthode non statique **valueOf()**.

p. 11 Solution n°12

Voici un code qui fonctionne :


```
1 const prix = new Number(678);
2
3 function verifPrice(pr) {
4   if(typeof pr == "object" && Number.isInteger(pr.valueOf()) && pr.valueOf() >= 500 &&
5     pr.valueOf() <= 2000) {
6     console.log("Le prix est valide");
7   }
8   else {
9     console.log("Le prix n'est pas valide");
10  }
11 }
12 verifPrice(prix);
```

Exercice p. 11 Solution n°13

Question 1

Quelle méthode permet de vérifier si un objet **Number** stocke un nombre entier ?


- ☒ `isInteger()`
- ☐ `isNaN()`
- ☐ `toFixed()`

 Nous pouvons utiliser **isInteger()** avec la syntaxe : **Number.isInteger(*nombre*)**. C'est une méthode statique, c'est pour cela qu'on l'appelle en passant par **Number**.

Question 2

Quelle propriété contient la valeur maximale qu'un objet **Number** peut stocker ?


- ☒ `Number.MAX_VALUE`
- ☐ `Number.MIN_VALUE`
- ☐ `Number.isFinite()`

 La constante **Number.MAX_VALUE** correspond au nombre maximal que l'on peut stocker dans un objet **Number**.

Question 3

Quelle méthode permet de convertir un objet **Number** en une chaîne de caractères ?


- ☒ `toString()`
- ☐ `valueOf()`
- ☐ `toFixed()`

 La méthode **toString()** permet de renvoyer une chaîne de caractères contenant la valeur du nombre d'un objet **Number**. Elle convertit donc la valeur de retour de **valueOf()** en chaîne de caractères.

Question 4

Quelle est la méthode qui permet de récupérer l'écriture exponentielle d'un nombre ?

- ☐ `toString()`
- ☒ `toExponential()`
- ☐ `toFixed()`
- ☐ `toFixed()`

 La méthode **toExponential()** permet de renvoyer le nombre stocké dans l'objet **Number** avec l'écriture exponentielle.

Question 5

Quelle est la méthode statique qui permet de vérifier uniquement si un nombre est fini ou non ?

- ☒ `isFinite()`
- ☐ `isNaN()`
- ☐ `isInteger()`

Q La méthode **isFinite()** permet de renvoyer un booléen afin de vérifier si un nombre est fini ou non. C'est une méthode statique, nous pouvons donc l'appeler avec la syntaxe : **Number.isFinite(*nombre*)**. Attention, c'est un type primitif **Number** qui doit être passé comme argument, et non un objet **Number**.