

Module 3 (atelier)

Types de valeurs

Objectifs

Les objectifs de cet atelier sont les suivants :

- découvrir les types de valeurs (primitives) manipulables en Javascript ;
- voir comment écrire ces valeurs (y compris les gabarits de chaînes de caractères) ;
- découvrir le fonctionnement des conversions automatiques et (dans une certaine mesure) savoir les prédire (ou les éviter) pour rédiger un code qui en tient compte.

Exercice 1 : les types de valeurs

Le but de cet exercice est d'observer comment Javascript traite différentes valeurs primitives : la notion de « type » de données est-elle complètement absente en Javascript ?

Étape 1 : un langage non typé

Javascript est un langage dit « non typé ». En C et en Java, avant d'utiliser une variable, il est nécessaire de la « déclarer » en lui donnant un type. Par la suite, la variable en question ne pourra jamais contenir que des valeurs de ce type.

Ainsi, le code

```
int monEntier;  
monEntier = "Bonjour";
```

produira une erreur de type en Java.

Ce n'est pas le cas en Javascript, où il n'est pas nécessaire de déclarer le type des variables et où une variable peut contenir des valeurs de types différents au fil de l'exécution (même si, en pratique, ce fonctionnement est à proscrire car il n'est guère compatible avec les principes du Clean Code).

Pour vous en convaincre, exécutez une à une les lignes suivantes dans la console de Firefox.

```
let x = "Bonjour";  
x; // pour obtenir la valeur actuelle de x  
x = 127;  
x;
```

Étape 2 : non typé = pas de type ?

Ceci dit, ce n'est pas parce qu'une variable n'est pas déclarée comme étant d'un type donné que son contenu, lui, n'a pas de type.

À nouveau, entrez les lignes suivantes une à une dans la console (ne répétez par le « let » si vous tapez ces lignes à la suite des précédentes).

```
x = 5;
x;           // pour afficher la valeur actuelle de x
x += 2;
x;           // pour afficher la valeur actuelle de x
x += " nains";
x;           // pour afficher la valeur actuelle de x
x += 2;
x;           // pour afficher la valeur actuelle de x
```

Observez l'effet des deux instructions « x += 2; ».

Dans le premier cas, on a ajouté l'entier 2 à la valeur de x, qui était l'entier 5. Dans le second cas, on a ajouté le caractère 2 à la valeur de x, qui était la chaîne « 7 nains ».

Ainsi, même si la variable « x » n'a pas de type associé, Javascript retient tout de même le type de son contenu. Et, en fonction du type du contenu de « x », l'instruction « x += 2; » pourra avoir des effets différents.

Ainsi, plutôt que de parler de « langage non typé », il est sans doute plus juste de parler de « langage dynamiquement typé ». Cet autre manière de désigner ces langages montre bien que les types existent toujours... mais que le type d'une variable (plutôt, de son contenu), peut évoluer (c'est-à-dire est dynamique).

Étape 3 : le type (du contenu) d'une variable

Javascript inclut une opération qui permet d'obtenir le type d'une variable (ou, plutôt de son contenu), à un moment donné. Cette opération est « typeof ».

Pour voir comment elle fonctionne, entrez une à une les lignes suivantes.

```
let mot = "Javascript";
typeof mot;
mot == "Javascript";
typeof (mot == "Javascript");
```

Observez que l'opération « typeof » renvoie le type de son argument sous la forme d'une chaîne de caractères.

Vous devriez donc être capable de prévoir ce que va afficher la ligne suivante... comparez votre prédiction avec le résultat donné par Javascript !

```
typeof ((typeof mot == "Javascript") == "boolean");
```

Notez au passage que Javascript permet de comparer des chaînes de caractères en utilisant « == », ce qui n'est pas le cas en C ni en Java. Vous souvenez-vous comment comparer deux chaînes en C ? Et en Java ?

Étape 4 : les types en Javascript

Vous savez donc que Javascript garde trace du type des données contenues dans ses variables. Reste à voir quels sont les types que Javascript permet d'utiliser.

Pour ce faire, vous allez affecter diverses valeurs à la variable x puis, à chaque fois, vous entrerez les quatre lignes suivantes dans la console Firefox. Ces quatre lignes afficheront (1) la valeur de x, pour vérification, (2) le type de cette valeur et (3, 4) la manière dont cette valeur est affichée sous forme de chaîne de caractères dans une fenêtre pop-up ou dans la console.

```
x; // Valeur de x
console.log("Type = " + typeof x); // Type de x
alert("x = " + x); // Écriture de x (pop-up)
console.log("x = " + x); // Écriture de x (console)
```

Étape 5 : premiers types

Commençons par des valeurs « familières » : des chaînes de caractères et des booléens.

Entrez la ligne

```
let x = "Hello";
```

puis reprenez les quatre lignes citées ci-dessus.

Faites de même avec

```
let x = true;
```

puis avec

```
let x = false;
```


Note sur le « let ». Si vous entrez toutes les lignes dans une même session, il ne faut pas répéter le « let » à chaque fois. Seul le premier (indiquant « voici une nouvelle variable ») est nécessaire. Plus à ce sujet dans le point sur la théorie qui suivra cet atelier.

Étape 6 : d'autres types

Réalisez les mêmes opérations pour les valeurs suivantes.

```
x = 3;
x = 7.18;
x = 3E-46;
```

Les langages C et Java utilisent (entre autres) les types short, int, long, float, double.

Qu'en est-il en Javascript ? 

Que produit la ligne suivante ? 

```
typeof 3.14 == typeof 77;
```

Étape 7 : plus loin dans les types

Jusqu'ici, vous devriez avoir déjà découvert (au moins) 3 types. Et si on n'affectait pas de valeur du tout ? Comment Javascript traite-t-il ce cas-là ?

Pour obtenir une réponse, vous allez devoir soit ouvrir une nouvelle session (c'est-à-dire ouvrir un nouvel onglet sur Firefox et travailler dans la console de ce nouvel onglet) ou alors utiliser un autre nom de variable (vu que, à ce moment-ci, x possède déjà une valeur).

Quelle que soit l'option choisie, déclarez une variable via une instruction du type

```
|| let x;
```

sans initialisation, puis exécutez les quatre lignes données ci-dessus.

Observez le résultat. Vous venez de découvrir un 4^e type de valeurs... pour lequel il n'existe qu'une seule valeur possible.

Ce type de valeurs et cette valeur portent le même nom : « undefined ».

Étape 8 : d'autres types

Léger spoiler... il existe d'autres types de valeurs qui peuvent être contenues dans une variable. Histoire d'en découvrir certains (c'est normal si vous ne comprenez pas tous les détails à ce moment-ci), effectuez les mêmes tests que ci-dessus après chacune des affectations suivantes.

```
|| x = function (z) { return z + 4 };  
|| x = [1, 2, 3];  
|| x = { nom : "Javascript", typé : false };  
|| x = { valeur : 17, toString () { return this.valeur; } };
```

Exercice 2 : les types primitifs

Javascript possède trois types dits « primitifs ». Parmi les types que vous avez identifiés ci-dessus, il s'agit de : Number (pour les valeurs numériques), String (pour les chaînes de caractères) et Boolean (pour les booléens).

Ces types sont dits « primitifs » parce que, comme en Java, ils fonctionnent par valeur plutôt que par référence. À l'inverse, les objets (comme ceux que vous avez identifiés dans la dernière étape de l'exercice 1), eux, fonctionnent plutôt par référence.

Le but de cet exercice est de voir quelles valeurs se trouvent dans chacun de ces types et comment on peut écrire ces valeurs.

Étape 1 : du côté des nombres

L'écriture des valeurs numériques en Javascript suit les mêmes règles que dans la plupart des autres langages. Cependant, le type « Number » comporte trois valeurs supplémentaires.

Pour les découvrir, exécutez les instructions suivantes une par une et observez les résultats.

```
|| function f(x) { return 12/x; }  
|| f(4);  
|| f(1);  
|| f(0.001);  
|| // Mathématiquement, que se passe-t-il si on calcule f(x) quand  
|| // x se rapproche de plus en plus de zéro ?  
|| f(0);
```

La seconde valeur spéciale de type « Number » est similaire à la précédente, mais correspond à une autre extrémité. Évaluez par exemple dans la console l'expression

```
|| -5/0;
```

Quant à la troisième valeur, on peut l'obtenir en évaluant

```
|| 0/0;
```

Le mot « NaN » est en fait l'abréviation de « Not a Number ».

Les deux premières valeurs numériques spéciales ne sont utilisées que dans le cas de calculs mathématiques spécifiques (comme la division d'un nombre non nul par zéro ou pour exprimer un dépassement de capacité).


La troisième valeur spéciale, NaN, par contre, correspond à des calculs qui n'ont pas de solution mathématique (comme 0/0) mais possède aussi une autre utilité, que vous découvrirez dans l'exercice suivant (sur les conversions).

Étape 2 : du côté des booléens


Il n'y a pas grand-chose à dire sur les booléens en Javascript... Ils s'écrivent « true » et « false ». Comme en Java, il s'agit de valeurs à part entière (contrairement au C où elles sont automatiquement traduites en entiers, généralement 0 et 1).

Quatre rappels de Clean Code relatifs à l'utilisation des booléens...


(1) Une variable booléenne seule suffit comme condition (dans un « if » ou dans une boucle par exemple). Si on veut tester que la variable « test » est vraie, il est inutile d'écrire

```
|| if (test == true) ...  // CODE À CORRIGER !!!
```


(2) Cette remarque est également valable dans le cas où la condition est composée (avec des « et » et des « ou » logiques). Un autre bout de code à ne pas reproduire :

```
|| if (cote >= 10 && aSoudoyéLeProf == true) ...  // CODE À CORRIGER !!!
```

(3) Si on veut tester qu'une variable booléenne est fausse, il vaut mieux utiliser la négation logique que de la comparer à « false ». Troisième bout de code à corriger :

```
|| if ( codevalide == false) ... // CODE À CORRIGER !!!
```

(4) Finalement, si on veut placer dans un booléen le résultat d'un test, on peut le faire en une seule affectation. Quatrième bout de code qui pique terriblement aux yeux :

```
|| if (age >= 18) 
||     majeur = true;
|| else
||     majeur = false; // CODE À CORRIGER !!!
```

Prenez le temps de réécrire chacun des quatre bouts de code moches présentés ci-dessus. N'oubliez pas que vous serez évalués non seulement sur le Javascript mais aussi sur la qualité de votre code (clean code et algorithmes utilisés) !

Étape 3 : du côté des chaînes de caractères

Contrairement au Java, où les chaînes de caractères sont traitées comme des objets, Javascript les considère comme des données primitives.

C'est pour cela qu'on peut par exemple les comparer en utilisant simplement « == ». En Javascript, le test « s1 == s2 » compare bien le contenu des deux chaînes et pas simplement leur adresse.

En Javascript, on peut écrire des chaînes de caractères en les entourant soit de guillemets soit d'apostrophes. Il faut cependant être cohérent (si on commence avec un guillemet, il faut finir avec un guillemet... même chose avec des apostrophes).

```
|| let salut = "Bon" + 'jour';
```

À l'intérieur d'une chaîne de caractères, on peut utiliser des « caractères échappés ». Il s'agit par exemple de \t (tabulation), \n (retour à la ligne) et \\ (pour un backslash). Si la chaîne est entourée de guillemets, pour y inclure un guillemet, il faut aussi l'échapper (\"). Idem si on veut inclure une apostrophe dans une chaîne encadrée par des apostrophes.

Par contre, pour écrire un guillemet dans une chaîne encadrée par des apostrophes, inutile de l'échapper. (Et vice versa).

Faites apparaître chacune des lignes ci-dessous (une par une) dans une fenêtre pop-up en utilisant l'instruction « alert (s); », où la variable s contiendra une chaîne de caractères que vous aurez définie. Choisissez judicieusement entre des guillemets et des apostrophes.

```
1 deux 3 // les espacements sont des tabulations
Aujourd'hui
ces "bons" vieux zigzags \\\/
t'es bien "gentil"
<li class='important'>C:\Users\etu12345</li>
```

Comme le HTML lui aussi permet d'utiliser des guillemets ou des apostrophes indifféremment, cette liberté permet de simplifier l'écriture en évitant les échappements autant que possible.

Étape 4 : gabarits de chaînes de caractères

Dans sa nouvelle version (ES6), Javascript autorise également des « gabarits » ou « template literals ». Il s'agit de chaînes de caractères dont certaines parties sont « calculées » : leur valeur provient soit directement de variables soit d'expressions.

Dans un premier temps, écrivez une fonction Javascript afficheProduit qui reçoit deux arguments (disons nb1 et nb2) et qui affiche dans la console (via console.log) ou dans une fenêtre popup (via alert) une chaîne de caractères indiquant « Le produit de <nb1> et de <nb2> vaut <produit> ». »

Ainsi, l'appel afficheProduit(3,7) devra afficher

```
|| Le produit de 3 et de 7 vaut 21.
```

Une fois la fonction définie et testée, passez à l'étape suivante.

Étape 5

Dans la fonction `afficheProduit`, vous avez peut-être défini une variable locale « produit » ou utilisé directement l'expression « `nb1 * nb2` » dans l'affichage. Mais, dans tous les cas, vous avez sans doute concaténé plusieurs petits bouts pour faire une chaîne de caractères :

```
|| "Le produit de " + nb1 + " et de " + nb2 + " vaut " + (nb1*nb2) + "."
```

Les « gabarits » permettent d'inclure toutes ces expressions (`nb1`, `nb2` ainsi que `nb1 * nb2`) directement dans la chaîne de caractères.

Dans ce cas-ci, voici la syntaxe à utiliser :

```
|| `Le produit de ${nb1} et de ${nb2} vaut ${nb1 * nb2}.`
```

Remarquez que

- le gabarit est encadré par des « apostrophes inverses » (vous pouvez les obtenir en laissant la touche ALT enfoncée et en tapant 96 sur le clavier numérique) ;
- chaque partie calculée est précédée de « \$ » et encadrée d'accolades.

Utilisez cette syntaxe et testez qu'elle produit bien le résultat escompté.

Étape 6

Dans l'exercice 5 du laboratoire introductif, vous avez sans doute défini une fonction `cellule(nb)` qui produit le code HTML

```
|| <td class="positif">nombre</td>
```

si le nombre donné était positif ou nul, et

```
|| <td class="negatif">nombre</td>
```

si le nombre donné était négatif.

Recodez cette fonction en utilisant un gabarit, sous la forme suivante.

```
|| function cellule (nb) {  
||   return ` ... `;  
|| }
```

Conseil. Utilisez une expression conditionnelle (« ... ? ... : ... »)

Exercice 3 : conversions

Considérons l'instruction suivante, qui pourrait être du C, du Java ou du Javascript.

```
|| res = x * y;
```

S'il s'agit de Java et que les variables `x` et `y` ont été déclarées via « `int x` » et « `String y` », une erreur sera produite : les types ne sont pas corrects. On ne peut pas multiplier un entier par une chaîne de caractères...

Mais que se passe-t-il en Javascript, où les variables ne sont pas associées à un type déterminé ? Les concepteurs du Javascript auraient pu choisir d'interrompre l'exécution si les contenus n'étaient pas compatibles... mais ils ont choisi une autre voie.

Dans la majorité des cas, Javascript va tenter de convertir automatiquement les données en valeurs qui peuvent être combinées. Dans le cas de l'exemple ci-dessus, comme il s'agit d'une multiplication, des nombres sont nécessaires. Les valeurs de x et de y vont donc être converties en nombres.

Pour vous en convaincre, évaluez la ligne suivante dans la console.

```
17 * "10";
```

Le résultat devrait être 170 : le 17 est bien numérique et la chaîne de caractères est convertie en le nombre 10.

Le but de cet exercice est de déterminer comment les valeurs sont converties.

Étape 1 : conversion en nombres

Les mots « Number », « String » et « Boolean » désignent non seulement trois types primitifs mais aussi des fonctions qui permettent de convertir une valeur dans le type en question. Ainsi,

```
Number("10");
```

donne le résultat de la conversion de la chaîne "10" en nombre.

Complétez le tableau suivant, qui indique comment diverses valeurs sont converties en nombres en observant le résultat donné par la fonction Number lorsqu'on l'applique aux exemples cités.

Conversion en nombres (via Number)		
Type de départ	Exemple(s)	Résultat
Nombre	17	17 (la valeur est inchangée)
Booléen	true	1
	false	0
Chaîne	"" (chaîne vide)	0
	" " (espaces)	0
	"14.2"	14.2
	" -18 "	-18
	"5 doigts"	NaN
	"pas un nombre"	NaN
Undefined	undefined	NaN

Notez qu'ici, la valeur numérique « NaN » prend tout son sens...

Sur base des exemples du tableau, prévoyez le résultat des expressions suivantes. Vérifiez ensuite vos prédictions dans la console.


```

false * "35" 0
"254" / "" Infinity
true * "-3" -3
"3 croissants" * "1 Euro par croissant" NaN
false * "true" NaN

```

Note importante. On aborde ce genre d'expressions pour mettre en évidence une partie des mécanismes que Javascript utilise automatiquement lors de l'exécution et pour montrer que l'aspect « non typé » d'un langage implique toute une série de conséquences auxquelles on ne pense pas forcément tout de suite (comme des règles de conversion relativement complexes). Il va de soi qu'il faut éviter autant que possible de se servir de ces particularités de Javascript lorsqu'on rédige du code. Un programme propre (clean code) effectuera des conversions explicites et s'assurera que les valeurs ont le bon type avant d'effectuer des opérations.

Étape 2 : conversion en booléen

En utilisant la fonction « Boolean », déterminez comment les valeurs citées ci-dessous sont converties.

Conversion en booléens (via Boolean)		
Type de départ	Exemple(s)	Résultat
Nombre	17	true
	-13.5	true
	0	false
	NaN	false
Booléen	true	true (inchangé)
	false	false (inchangé)
Chaîne	"" (chaîne vide)	false
	" " (espaces)	true
	"0"	true
	"true"	true
Undefined	undefined	false

Sur base des observations du tableau, prévoyez le résultat des expressions suivantes. Vérifiez ensuite vos prédictions dans la console.

```

"ok" ? 10 : -10 10
"" ? 10 : -10 -10
undefined ? 10 : -10 -10

```

Étape 3 : conversion en chaîne

Finalement, la fonction « String » permet de convertir une valeur en chaîne de caractères. De manière générale, on peut décrire String(x) comme l'écriture standard de la valeur x.

Ainsi, pour un nombre, on obtiendra la chaîne qui correspond à l'écriture du nombre. Pour undefined, on obtiendra la chaîne "undefined". Les booléens, quant à eux, seront transformés en les chaînes "true" et "false".

Effectuez quelques tests de la fonction `String` pour vous assurer que les informations du paragraphe précédent sont bien correctes. Ensuite, prévoyez le résultat des expressions suivantes puis vérifiez vos prédictions en utilisant la console.

```
String(true) * 1 NaN
Boolean(String(false)) true car convertit en "false" et ensuite false
est une chaîne de caractères non vide donc TRUE
```

Exercice 4 : Opérations et conversions

Savoir comment Javascript convertit les valeurs en d'autres types est une chose... mais, pour être complet, il faudrait également savoir quand Javascript effectue une conversion, et comment le type cible est choisi.

Dans le cas de l'exemple introductif

```
res = x * y;
```

où l'opération (la multiplication) ne peut s'appliquer que sur des nombres, il n'y a pas de doutes : Javascript va convertir `x` et `y` en nombres.

Mais ce n'est pas toujours aussi clair...

Note. Pour comprendre cette partie de l'atelier, vous devez maîtriser les mots de vocabulaire suivants. Si ce n'est pas le cas, cherchez une définition sur le web (par exemple sur Wikipédia).

- Surcharge (programmation informatique),
- Surcharge d'opérateurs
- Opérateur
- Opérande paramètres
- Ordre lexicographique par ordre Alphabétique par exemple ou encore , 100 < 47 selon l'ordre lexicographique

Étape 1 : le cas de « + »

L'opération « + » est une opération dite « surchargée ». En informatique, cela signifie qu'un même symbole est utilisé pour représenter des choses différentes. On retrouve cette même notion en Java quand on décide de « surcharger » une méthode pour lui permettre d'accepter différents types d'arguments : un même nom de méthode correspond alors à des codes différents.

Le symbole « + » peut désigner à la fois une addition (sur des nombres) et une concaténation (sur des chaînes de caractères). Voici la règle utilisée par Javascript pour savoir gérer cette opération en fonction du type des opérandes.

Évaluation de `val1 + val2` en Javascript

- Si aucune des deux valeurs n'est une chaîne de caractères :
convertir les deux valeurs en nombres puis les additionner
- Sinon (au moins une des valeurs est une chaîne de caractères) :
convertir les deux valeurs en chaînes de caractères puis les concaténer

Par exemple, dans la ligne Javascript

```
|| 3 + "2"
```

comme au moins une des valeurs est une chaîne de caractères (la seconde opérande), on convertit les deux valeurs en chaînes et on les concatène. Si vous vous reportez à l'exercice précédent, vous verrez qu'un nombre (comme 3) converti en chaîne de caractères devient l'écriture de ce nombre, c'est-à-dire la chaîne "3". Le résultat de l'expression ci-dessus est donc la chaîne "32" (concaténation des deux).

Par contre, la ligne Javascript

```
|| 7 + true
```

ne contient aucune chaîne de caractères. Les deux opérandes sont donc converties en nombres. Une fois de plus, en suivant les règles établies dans l'exercice précédent, on trouve que `Number(7) = 7` et `Number(true) = 1`. Le résultat final est donc $7 + 1 = 8$.

En utilisant toutes les connaissances accumulées jusqu'ici, prévoyez le résultat de chacune des expressions suivantes puis vérifiez vos prédictions en utilisant la console.

```
"42" + "1" 421
42 + "1" 421
42 - "1" 41
"42" + true 43
"42" + (true + 0) 421 car au moins une chaîne de caractère et true converti
42 + (true + "0") 42true0 en number
42 + true 43
42 + true + "1" 431
(true + (2 * true)) * ("35" / 7) 3 * 5 = 15
(true + 2 * "5") 11
```

Étape 2 : le cas de "=="

Comme en Java et en C, le symbole « == » permet de comparer deux valeurs. Dans le cas des valeurs primitives (nombres, booléens et chaînes), on compare bel et bien les valeurs ; dans le cas des objets, on compare plutôt l'adresse.

Attention... en Java, les chaînes sont considérées comme des objets. Par contre, en Javascript, il s'agit bien de valeurs primitives (qui peuvent donc être comparées avec ==).

Si on compare deux valeurs de même type... la comparaison teste simplement l'égalité des valeurs. Mais que se passe-t-il si les opérandes de « == » sont de types différents ? Il faut alors convertir les opérandes dans un même type puis comparer les valeurs obtenues.

Plus précisément, voici la règle utilisée par Javascript. Attention... on arrive déjà ici à un niveau de difficulté légèrement supérieur au cas de « + » !

Évaluation de $val_1 == val_2$ en Javascript

- Si les deux opérandes sont de même type, on effectue une comparaison de valeurs.
Exception : la valeur numérique spéciale NaN n'est égale à personne, pas même à elle-même ! Le test `NaN == NaN` donne donc... false !
- Sinon, si parmi les opérandes, on trouve au moins une fois la valeur undefined ou null, si le test est `null == undefined` ou `undefined == null`, la réponse est true ; dans les autres cas, la réponse est false.
- Sinon (c'est-à-dire types différents mais pas de undefined ni de null), on convertit les deux opérandes en nombres et on compare les valeurs.

Par exemple, l'expression Javascript

```
|| "4" == true
```

possède des opérandes de types différents (string et booléen). Comme il n'y a ni null ni undefined, on **les convertit toutes les deux en nombres**. D'après les règles de l'exercice précédent, `Number("4") = 4` et `Number(true) = 1`. Au final, 4 et 1 sont des valeurs différentes, donc la réponse est... false.

Dans l'expression Javascript

```
|| 5 == "5"
```

on trouve également des opérandes de types différents. À nouveau, on convertit en nombres : `Number(5) = 5` et `Number("5") = 5`. Comme les deux valeurs numériques obtenues sont égales, l'expression s'évalue à true.

Notez que l'opérateur « != » utilise les mêmes règles, mais renvoie le résultat opposé.

En utilisant toutes les connaissances accumulées jusqu'ici, prévoyez le résultat de chacune des expressions suivantes puis vérifiez vos prédictions en utilisant la console.

```
'2' == 2    true
"  " == 0   true
0 == ""    true
"" == "0"  false
0 == "0"   true
false == undefined true
true == null false
false == null false
1 == true  true
"0" != false false
421 == ("42" + 1) true
(25 == "25") + 9 10
"1" == (0 + true) true
(("14" * 2) == ("20" + 8)) false
true * "0" === false + "1" false
```

Étape 3 : l'opérateur « === »

Dans certains cas, on peut avoir envie de tester l'égalité de deux valeurs sans tenir compte des conversions automatiques. À cette fin, Javascript introduit l'opérateur « === » (trois symboles « égal »).

Sa signification est la suivante :

Évaluation de $val_1 === val_2$ en Javascript

- Si les deux opérandes sont de types différents, la réponse est false.
- Sinon (si les deux opérandes sont de même type), le résultat est le même que pour l'opérateur ==.

L'opérateur « !== » correspond à la négation de « === ».

En utilisant toutes les connaissances accumulées jusqu'ici, prévoyez le résultat de chacune des expressions suivantes puis vérifiez vos prédictions en utilisant la console.

```
"33" === 33 false
null === undefined false
NaN !== NaN true
```

Étape 4 : les opérateurs de comparaison <, <=, >, >=

Dernier cas abordé dans le cadre de cet atelier : celui des opérateurs d'ordre. À nouveau, si les opérandes sont de même type, la comparaison a une signification claire. Par contre, si elles ont des types différents, cela peut être plus compliqué.

Considérez l'exemple suivant, basé sur les nombres 123 et 9.

On a naturellement que

```
9 < 123 // comparaison de nombres
"123" < "9" // comparaison de chaînes
```

mais que se passe-t-il si on compare "9" et 123, ou encore 9 et "123" ?

Voici la règle qui permet de répondre à cette question.

Évaluation de $val_1 < val_2$ en Javascript

- Si les deux opérandes sont des chaînes de caractères, on utilise l'ordre lexicographique.
- Sinon (au moins une opérande qui n'est pas une chaîne), on convertit les deux opérandes en nombres et on compare les valeurs obtenues.

Ainsi, lorsqu'on compare "9" et 123 ou bien 9 et "123", les deux sont convertis en nombres et la comparaison s'effectue donc sur 9 et 123.

En utilisant toutes les connaissances accumulées jusqu'ici, prévoyez le résultat de chacune des expressions suivantes puis vérifiez vos prédictions en utilisant la console.

```
31 < "274" true
"147" < "25" true
"150" < 99 false
420 < ("42" + true * "1")true
"43" < "421" false
"0" < ("0" * 421 == false) true
```

Étape 5 : remarques finales

Tout comme les règles de conversion (voir l'exercice précédent), les définitions des opérateurs abordés dans cet exercice sont plutôt complexes, voire parfois un peu obscures. L'objectif de cet atelier n'est absolument pas de vous inciter à retenir tous les détails de ces règles par cœur ! Au lieu de cela, il s'agit plutôt de vous mettre au courant de leur existence et de s'assurer que vous êtes capables de comprendre ces règles.

Pourquoi voir ces règles dans un cours de Javascript ?

Comme Javascript est un langage non typé et que la plupart des opérateurs fournissent un résultat quel que soit le type de leurs opérands, de nombreuses erreurs de code passent sous silence. En Java ou en C, si vous tentiez de multiplier un entier et une chaîne de caractères, le compilateur indiquerait une erreur. En Javascript, aucune erreur ne sera mise en évidence, et le programme s'exécutera.

Si vous écrivez un programme Javascript et que celui-ci s'exécute sans erreur mais produit un résultat inattendu, ayez donc le réflexe de penser qu'il s'agit peut-être d'un problème de conversion implicite et ré-examinez votre code avec cette idée en tête.

Faut-il tirer parti de ces règles dans son code ?

Si on connaît sur le bout des doigts les définitions des opérateurs et les règles de conversion, on peut se baser sur elles pour écrire un code qui semble simple mais qui peut accomplir beaucoup de choses. Prenez par exemple le code « `x + y` » qui semble tout simple et facile à comprendre et rappelez-vous tout ce qui se cache derrière ces 3 symboles (test de types, conversions implicites, addition, concaténation...).

Le but premier du « clean code » est de rendre son code facilement lisible. Cela implique sans doute de ne pas abuser de particularités propres à un langage de programmation donné... En effet, bon nombre de programmeurs jonglent avec trois, quatre langages de programmation ou plus... Et on ne peut pas leur demander de connaître sur le bout des doigts les spécificités de chacun d'eux. Pour rendre un code lisible, il vaut donc mieux éviter de trop se reposer sur ces aspects « plus obscurs » de Javascript.

En pratique, cela peut signifier qu'il vaut mieux ajouter des conversions explicites pour rendre le code plus facile à lire. Par exemple, dans le test « `x == 3` », la valeur de `x` va être convertie en un nombre avant d'être comparée avec 3. Le résultat sera donc le même que celui du test « `Number(x) === 3` »... mais cette dernière écriture a l'avantage d'être beaucoup plus facile à comprendre par un lecteur qui n'est pas forcément un expert en Javascript.