

Bonjour!

Je m'appelle Frédéric BISSON, je suis développeur et je travaille à Rouen (Normandie).

On va s'attarder sur ces types de données qu'on manipule fréquemment sans faire bien attention aux pièges qu'ils recèlent.



Nous allons parler de nombres (entiers, flottants), de dates et, gros morceaux, des chaînes de caractères.

Des pistes seront données à chaque fois pour tenter d'améliorer la situation.



Un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur.

Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça ce sont les attributs de la magie).

> DAVE SMALL ST MAGAZINE N°66 - 1992



Citation de Dave Small dans le ST Magazine n°66 de 1992 : un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas censé être obscure, bizarre et plein de pièges subtils (ça ce sont les attributs de la magie).



Nous dirons LOL.

Préambule

- Cette présentation s'adresse
 - aux développeuses/eurs débutant/es
 - aux développeuses/eurs confirmé/es
- C'est une mise en garde sur des faux-amis
 - vous les utilisez à chaque fois que vous développez
 - vous les manipulez sans y prendre garde
 - ils sont la source de nombreux bugs et failles de sécurité



Cette présentation s'adresse à tout développeur ou développeuse, qu'on soit débutant ou confirmé. La réalisation de cette présentation a d'ailleurs permis de mettre en lumière mes propres lacunes sur tous ces sujets.

C'est une mise en garde sur des faux-amis qu'on utilise jour après jour jusqu'à ne plus y prendre garde. Ils sont pourtant la source de nombreux bugs et faille de sécurité.

Des types que l'on croît connaître

- Même les types les plus utilisés sont souvent méconnus
 - nombres entiers ou à virgule
 - dates
 - chaînes de caractères
- Et il existe des types qui n'en sont pas
 - URL
 - adresses mail
 - chemins

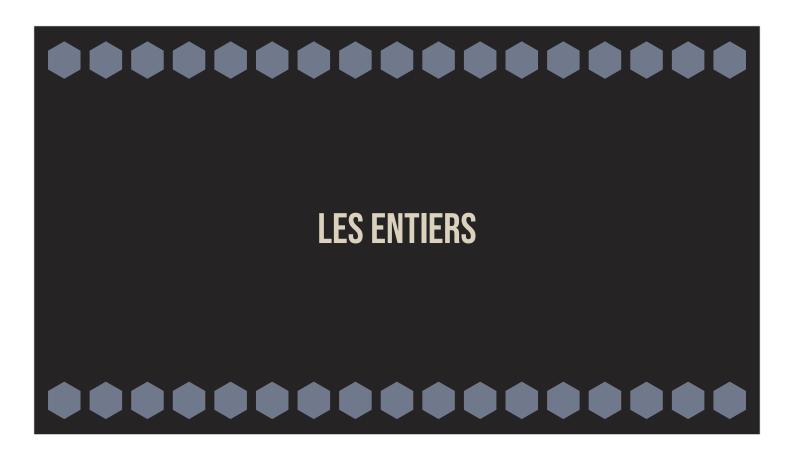


Ces types de données ce sont les nombres, les dates et les chaînes de caractères, les composants de base de la majorité des programmes.

La chaîne de caractères est probablement le type auquel il faut prêter le plus d'attention car il est utilisé à l'extrême pour échanger / conserver / manipuler d'autres types de données.



Commençons par le plus simple : les nombres.



Et le type qui existe depuis le début : les entiers.



- Les comportements ne sont pas les mêmes partout
 - encodage
 - différences avec les entiers vus en mathématiques
 - limites et comportement aux limites
 - résultat des opérateurs
- Les comportements évoluent
- Un entier n'est pas toujours un entier



Mais alors, qu'est-ce qui ne va pas avec les entiers?

Les comportements ne sont pas les mêmes partout, que ce soit au niveau de l'encodage (représentation physique), des différences avec les entiers qu'on a tous vus à l'école, les limites et le comportement aux limites et le résultat des opérateurs.

Et ces comportements évoluent!

Il faut de plus se méfier : des chiffres ne font pas toujours des nombres.

Un type de données natif et efficient

- Seul type supporté nativement par tous les processeurs
- La plus grande vitesse de traitement
 - instructions dédiées
 - instructions rapides
 - instructions SIMD
- Instructions les moins gourmandes en énergie
- · Encodage dépendant du processeur



Les entiers sont le seul type supporté nativement par tous les microprocesseurs, micro-contrôleurs etc.

Ils permettent la plus grande vitesse de calcul sur ordinateur grâce à des instructions dédiées, rapides, vectorielles, tout en étant le moins gourmand en énergie.

Leur représentation physique est dépendante du processeur : ordre des bits, signe, ordre des mots...

Des entiers bornés

- Nombre de bits couramment utilisés
 - **8** bits → **256** valeurs
 - **16** bits → **65 536** valeurs
 - 32 bits → 4 294 967 296 valeurs
 ~4 milliards de valeurs
 - 64 bits → 18 446 744 073 709 551 616 valeurs
 ~18 milliards de milliards de valeurs
- De nombreuses variantes
 - 10, 12, 15, 18, 20, 24, 36, 48, 80, 128 bits...



Première différence avec les entiers qu'on a tous vus à l'école : les entiers informatique sont bornés. Contrairement aux mathématiques, l'informatique est un domaine aux ressources limitées.

L'évolution de l'informatique et du matériel a abouti à des ordinateurs fonctionnant sur un nombre de bits multiple de 8 (8, 16, 32, 64, 128 bits).

Plus le nombre de bits est élevé, plus on va pouvoir coder de valeurs différentes.

Il ne faut cependant pas oublier qu'il existe de très nombreuses variantes, pour des raisons historiques, physiques, d'optimisation etc.



- Calcul multi-précision
 - n'existe pas nativement sur les processeurs
 - GNU MP souvent utilisé : Python, Haskell, Maple, Mathematica...
- Espace mémoire dépendant du nombre à coder
 - même les calculs simples entraînent des allocations mémoire
 - estimation complexe des besoins mémoire



Il existe aussi des entiers non bornés, ou presque. C'est ce que l'on appelle le calcul multi-précision.

Il n'existe pas nativement sur les processeurs.

S'il permet de faire des calculs sur des nombres monstrueux au prix de performances moindres par rapport aux entiers natifs, il entraîne de nouvelles problématiques pour le développement en environnement contraint. Chaque opération peut nécessiter une gestion mémoire complexe.



signé

[0..255]8 bits [-128..127]

16 bits [0..65 535] [-32 768..32 767]

32 bits [0..4 294 967 296] [-2 147 483 648..2 147 483 647]

Différence gérée par le programme!



2 types d'entiers cohabitent sur les architectures matérielles : les entiers positifs et les entiers relatifs. Le nombre de valeurs reste le même.

Les processeurs ne gèrent pas cette distinction, elle est de la responsabilité du programme.

La technique du complément à deux utilisée permet aux processeurs de manipuler ces deux types indistinctement.



Chaque langage dispose de ses propres variantes.

Le langage C créé à l'origine pour le développement de système d'exploitation portable est celui qui supporte la plus grande variété.

Les langages plus haut niveau actuels se limitent à quelques variantes, toutes signées, généralement basées sur la taille de mot maximum de la plateforme sur laquelle ils tournent.

JavaScript est un cas particulier avec ses entiers limités à 54 bits.

Pas d'entier natif en JavaScript

- Utilisation de nombres à virgule flottante
- Tout va bien tant que $-2^{53} \le x < 2^{53}$
- Rien ne va plus quand on sort de cet intervalle...

```
for (let i = 2**53; i < 2**53 + 2; i++);
très amusant à lancer dans la console du navigateur!
```

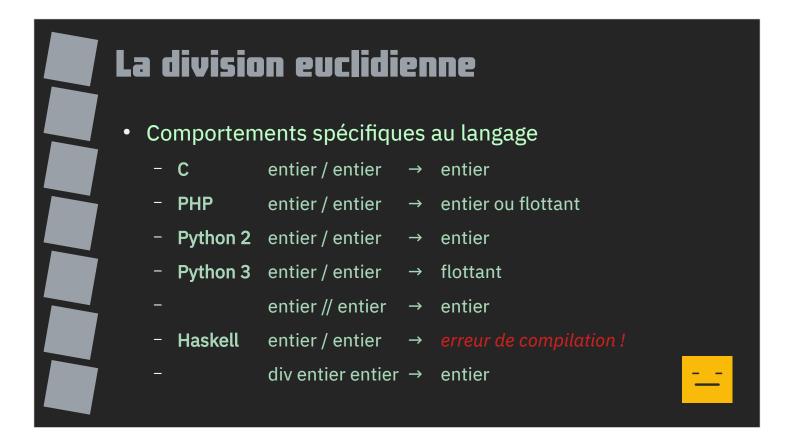


En fait, JavaScript ne sait pas manipuler nativement des entiers.

JavaScript ne sait manipuler que des nombres à virgule flottante. Les nombres à virgule flottante permettent de stocker des entiers sur 54 bits.

Au-delà de cette limite, une perte de précision se produit.

Les dernières évolutions de JavaScript disposent de bibliothèques natives pour palier au problème (par exemple BigInt)



Il y a division et division.

Mathématiquement, pour les entiers, il n'y a que la division euclidienne.

Mais en fonction de l'opérateur utilisé, chaque langage va opérer selon ses propres critères.

PHP pourra générer un flottant si la division ne tombe pas juste.

Haskell et son typage fort vous interdiront d'utiliser la division non euclidienne sur des entiers.

Python a évolué dans son comportement vis-à-vis de l'opérateur /.



La division par zéro est elle-même un cas particulier de traitement en fonction du langage.

C, Python et Haskell génèrent une exception (arrêt immédiat du programme), PHP retourne l'infini et émet un warning, JavaScript retourne silencieusement l'infini.

Tu me fais tourner la tête

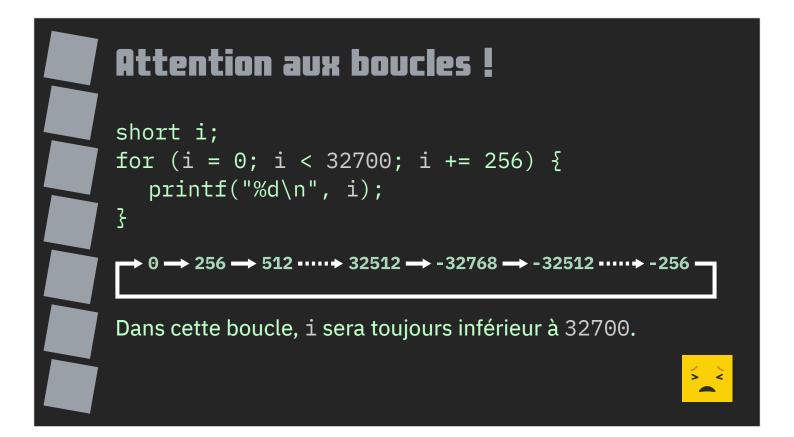
- Les entiers bornés fonctionnent comme des anneaux
 - entiers sur 8 bits → anneau Z/256Z
- Exemples
 - -255 + 1 = 0 pour des entiers non signés sur 8 bits
 - 127 + 1 = -128 pour des entiers signés sur 8 bits
- Attention!
 - a + b > a pour b > 0 n'est pas toujours vrai!



Les entiers bornés ne sont pas plafonnés mais fonctionnent comme des anneaux.

Pour des entiers non signés sur 8 bits, 255 + 1 = 0.

Cela a pour conséquence que des comparaisons parfaitement vraies en mathématique ne tiennent plus dans un langage donné. Par exemple a + b pour b supérieur à 0 n'est pas forcément supérieur à a.



Ce comportement doit être particulièrement anticipé lors de la création de boucles comme celle-ci.

En C, si on fait varier i, entier signé de 16 bits, de 0 à 32700 avec un incrément de 256, on obtient une boucle infinie car i sera toujours inférieur à 32700.

Des langages comme Python apportent une solution à ce type de problème avec la fonction range.

Note : le fait de passer à des entiers de taille supérieure (32 ou 64 bits) ne corrigent pas le problème, cela ne fait que le masquer.

Quand ça dépasse

- Comportements spécifiques au langage
 - C → fonctionne comme un anneau
 - PHP → integer devient double
 - Python 2 → int devient long int
 - Python 3 → int est en réalité long int
 - Haskell → fonctionne comme un anneau
 - JavaScript → perte de précision, plafonnement à Infinity



Là encore, chaque langage gère le dépassement à sa manière.

En cas de dépassement, PHP transforme votre entier en flottant, Python passe à des entiers à précision arbitraire, C et Haskell fonctionne comme des anneaux.

PHP et JavaScript aboutissant à des flottants « plafonneront » les résultats à l'infini.

Un entier qui n'en est pas un

- Chiffre ≠ nombre
 - numéro de sécurité sociale
 - numéro ou code secret de carte bleu
 - numéro de compte bancaire
 - etc.
- Les opérations mathématiques ne les concernent pas
 - addition, soustraction, multiplication, division...



Pour finir, il faut préciser que l'utilisation de chiffres n'implique pas l'utilisation d'un nombre.

Par exemple, un numéro de sécurité sociale n'est pas un nombre mais une suite de chiffre formant un identifiant. De même pour un code de carte bleu, un numéro de compte bancaire etc.

Il est en effet absurde d'envisager de multiplier deux numéros de sécurité sociale.



Alors, que faire?

Tout d'abord, replongez-vous dans les spécifications du langage que vous utilisez. Cette présentation ne révèle aucun secret, toutes les informations, pièges et comportements aux limites sont documentés.

Identifiez les cas aux limites et testez-les.

Identifiez les types de données adéquats pour votre usage.

Typez fortement, encapsuler afin de coller au plus près du comportement que vous attendez plutôt que de coller au comportement des types natifs.

Ne vous reposez pas sur des hypothèses ou intuitions.



Corsons un peu la difficulté avec les nombres à virgule flottante ou flottants.

IEEE 754, pour les gouverner tous

- Norme omniprésente (JavaScript, C, Python, Haskell...)
- Une norme qui évolue
 - **IEEE 754-1985**, norme initiale, base 2
 - **IEEE 854-1987**, base 10
 - **IEEE 754-2008** = IEEE 754-1985 + IEEE 854-1987
 - IEEE 754-2019, révision mineure
- IEEE 754 ≠ ℝ
- Chacun applique la norme à sa façon...



S'il y a bien une norme pour gérer les nombres à virgule flottante, c'est IEEE 754. Elle est omniprésente dans les langages, dans le matériel.

Elle date de 1985 et évolue de temps en temps. Les langages n'implémentent pas la totalité de cette norme.

Première mise en garde : IEEE 754 n'est pas l'ensemble des rationnels.

Approximations

- 0.1 + 0.2 != 0.3
- 0.1, 0.2 et 0.3 ne peuvent s'écrire en base 2

$$-0.1_{10} = 0.0\overline{0011}_{2} = 0.00011001100110011_{2}...$$

$$-0.2_{10} = 0.\overline{0011}_{2} = 0.001100110011001100110_{2}...$$

$$-0.3_{10} = 0.01\overline{0011}_{2} = 0.01001100110011001_{2}...$$

- 4 arrondis cumulés
 - 3 conversions (0.1, 0.2 et 0.3) en base 2
 - 1 addition de 0.1 et 0.2



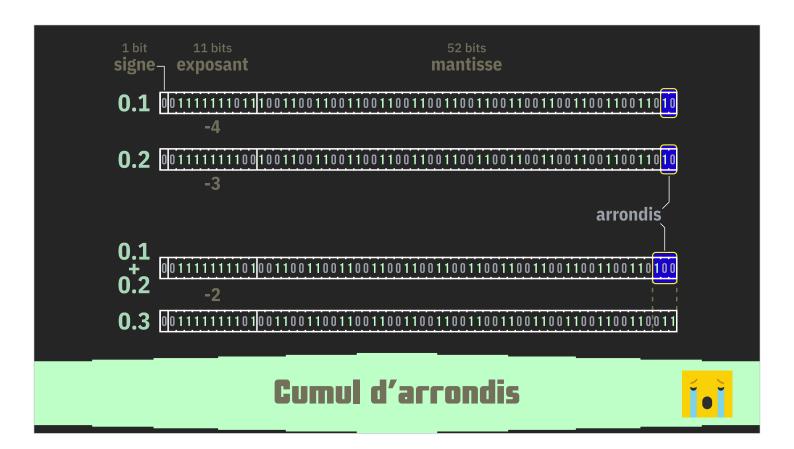
IEEE 754 est une approximation bourrée d'arrondis.

L'exemple tarte-à-la-crème est le fameux calcul 0.1 + 0.2 qui n'est pas égal à 0.3.

La norme travaille en base 2. Cela facilite l'implémentation et l'optimisation matérielle mais entraîne d'autres problèmes.

0.1 en base 10, c'est 0.1. En base 2, c'est 0.0001100110011 à l'infini. Même problème pour 0.2 et 0.3. Il est impossible d'écrire ces nombres en base 2!

Notre calcul 0.1 + 0.2 va donc cumuler 4 arrondis : 3 arrondis de conversion et 1 arrondi d'addition.



Physiquement, les flottants IEEE 754 sur 64 bits utilisent 1 bit de signe, 11 bits d'exposant et 52 bits de mantisse.

Le schéma montrent les bits sur lesquels l'arrondi a opéré.

Les vraies valeurs de 0.1 à 0.9



- 0.1000000000000000055511151231257827021181583404541015625
- 0.20000000000000011102230246251565404236316680908203125
- 0.29999999999999988897769753748434595763683319091796875
- 0.40000000000000002220446049250313080847263336181640625
- 0.5
- 0.599999999999997779553950749686919152736663818359375
- $\bullet \quad 0.6999999999999995559 \\ \underline{10790149937383830547332763671875}$
- 0.8000000000000000444089209850062616169452667236328125
- 0.9000000000000002220446049250313080847263336181640625

Même si votre langage vous permet d'afficher 0.1 quand vous lui indiquez 0.1, il passe obligatoirement par 2 phases d'arrondis : une à la conversion et une à les restitution.

Hormis 0.5, les nombres de 0.1 à 0.9 n'ont qu'une valeur approchée.

Dès qu'on travaille avec des flottants de façon répétée, il faut tenir compte de cette accumulation d'arrondis.

En JavaScript (mais fonctionne sur d'autres langages), additionner continuellement 0.3 ne donne que des arrondis même si le résultat final ne s'éloigne pas trop de celui attendu.

Notez que votre langage peut étendre la précision temporairement pendant les calculs (à 80 ou 128 bits), ce qui peut poser problème en cas de portabilité.

C'est tout autant valable pour votre tableur!



- Propriétés conservées
 - éléments neutres : 0 pour l'addition, 1 pour la multiplication
 - addition et multiplication sont commutatives
 - x x == 0
- Propriétés non conservées
 - multiplication n'est **pas distributive** par rapport à l'addition
 - addition et multiplication ne sont pas associatives
 - $\exists x \ x^*(1/x) != 1 ; \exists (a,b) \ a == b \land 1/a != 1/b$



Si certaines propriétés sont conservées, ce n'est pas le cas de toutes.

Les éléments neutres fonctionnent comme tels, l'addition et la multiplication sont commutatives (a+b=b+a, $a\times b=b\times a$) et x-x=0.

Mais la multiplication n'est pas distributive par rapport à l'addition : $a \times (b+c)$!= $a \times b + a \times c$.

L'addition et la multiplication ne sont pas associatives.

Multiplier un nombre par son inverse n'est pas toujours égal à 1.

L'addition n'est pas associative

$$(0.1 + 0.2) + (0.1 + 0.2) = 0.600000000000001$$

Véritable valeur: 0.600000000000000088817841970012523233890533447265625

$$0.1 + (0.2 + 0.1 + 0.2) = 0.6$$

Véritable valeur: 0.5999999999999997779553950749686919152736663818359375



Voici un exemple démontrant que l'addition n'est pas associative.

La différence reste cependant très faible et ce niveau de précision est généralement suffisant.

Les véritables valeurs physiques sont données sous les calculs.

La multiplication n'est pas associative

$$(0.1 * 0.2) * 0.3 = 0.006000000000000001$$

Véritable valeur: 0.006000000000000000992261828258733658003620803356170654296875

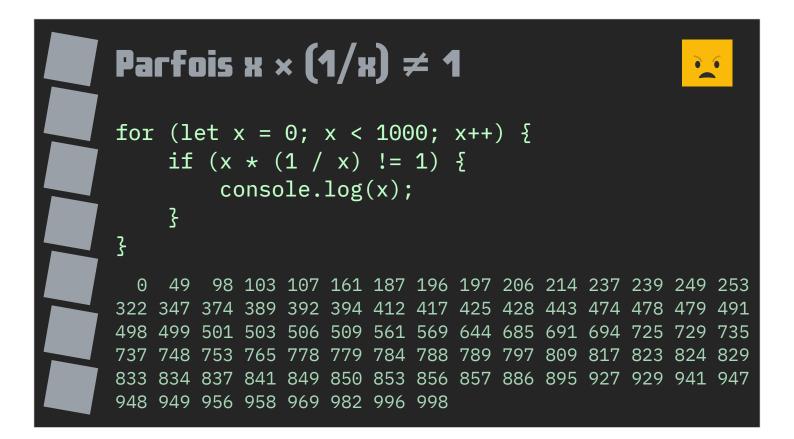
$$0.1 * (0.2 * 0.3) = 0.006$$

Véritable valeur: 0.006000000000000000012490009027033011079765856266021728515625



Et voici un exemple démontrant que la multiplication n'est pas associative non plus.

Les véritables valeurs physiques sont données sous les calculs.



Le problème d'arrondi se retrouve également quand on multiplie un nombre par son inverse. Mathématiquement cela devrait toujours donner 1.

Cette boucle en JavaScript montre tous les nombres entiers entre 0 et 1000 pour lesquels ce n'est pas vrai.

Encore une fois, la différence est faible.

Division par zéro Comportements spécifiques au langage - C flottant / 0 → inf flottant / 0 → INF PHP Warning: Division by zero PHP Python 2 flottant / 0 → ZeroDivisionError: floαt division by zero Python 3 flottant / 0 → ZeroDivisionError: floαt division by zero flottant / 0 → Infinity Haskell **JavaScript** flottant / 0 → Infinity



Selon la norme IEEE 754, la division par zéro est tout à fait valable et devrait normalement retourner l'infini ou moins l'infini.

Les langages appliquent généralement cette règle mais pas Python par exemple. PHP l'applique mais affiche quand même un warning.

NaN: pas toujours un nombre

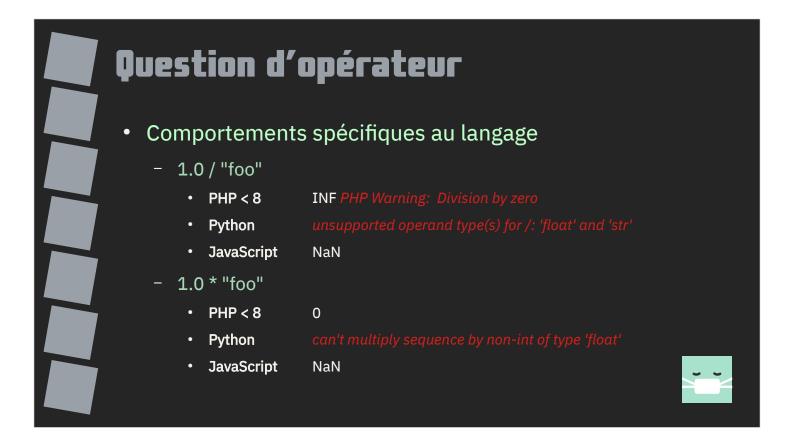


- Addition, soustraction : ∞ ∞
- Multiplication, division : $\infty \times 0$, $0 \div \infty$, $0 \div 0$, $\infty \div 0$
- Tout calcul avec NaN donne NaN



IEEE 754 apporte aussi la notion de nombre qui n'est pas un nombre : le NaN (ou Not A Number).

La norme supporte l'infini et quelques opérations. Les cas non déterminés conduisent à un NaN.



Les langages dynamiques apportent leurs petits problèmes. Un langage compilé n'autorisera normalement pas de mélange de types.

Mais des langages comme PHP, Python ou JavaScript déterminent le type des données manipulées au dernier moment.

Et le comportement en fonction de l'opérateur peut varier.

Petite subtilité : -0 et +0

- Égalité: -0 == 0
- Division

```
- 1 / -0 == -Infinity
```

- Infinity / -0 == -Infinity
- Addition

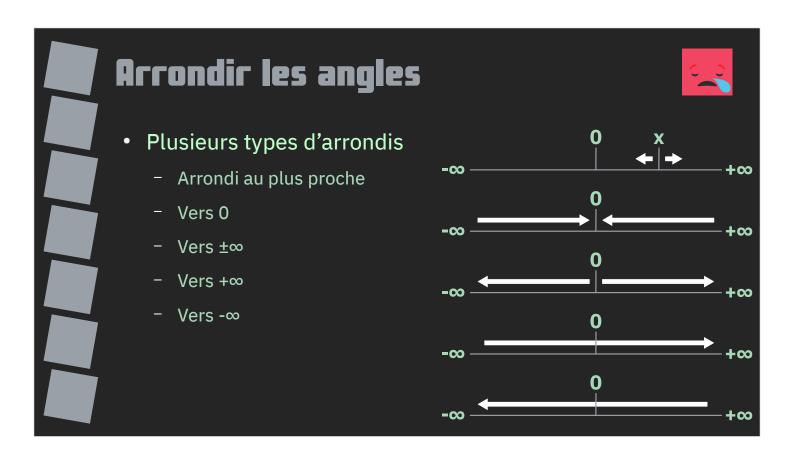
$$- -0 + -0 == -0$$



L'IEEE 754 implémente une subtilité qui n'existe pas en mathématique : le zéro signé.

Si -0 = 0 strictement, les opérations prennent en compte le signe du zéro.

Cela donnent des résultats cohérents pour la multiplication et la division mais pose question pour l'addition.



La norme IEEE 754 recense plusieurs types d'arrondis, plus ou moins supportés par les bibliothèques standards des différents langages.

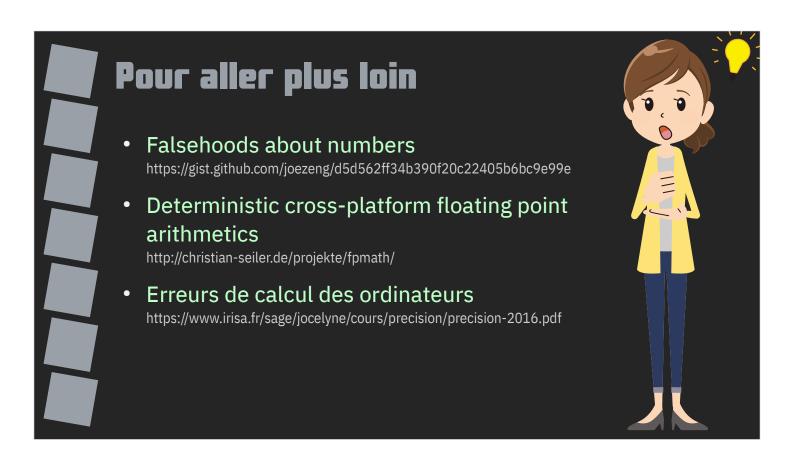


Alors, que faire?

Prendre conscience des différences, limites et comportements spécifiques aux flottants.

Testez les cas aux limites, y compris les cas bizarres avec l'infini ou NaN.

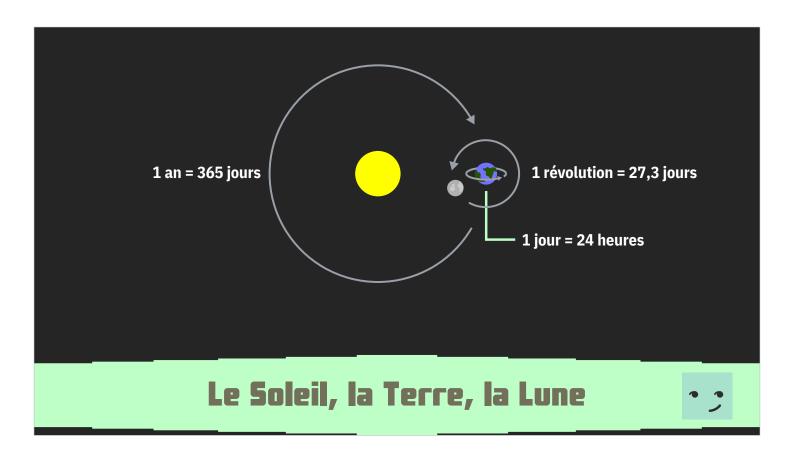
Et plus ou moins les mêmes recommandations que pour les entiers.



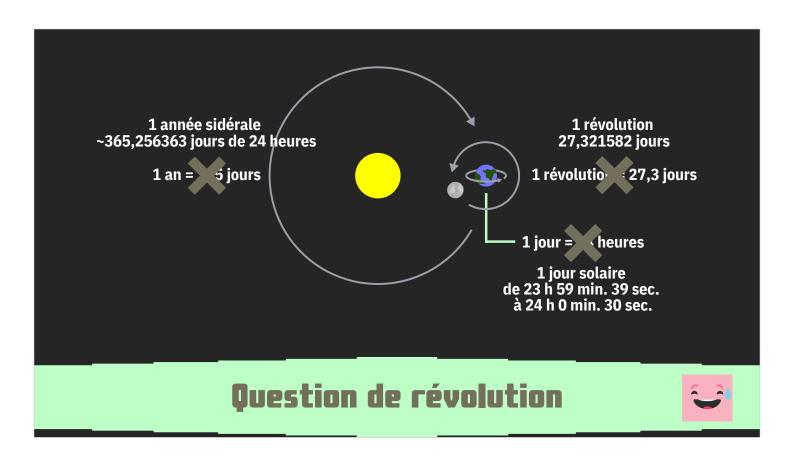
Voici quelques liens pour approfondir les problématiques de calculs avec les flottants.



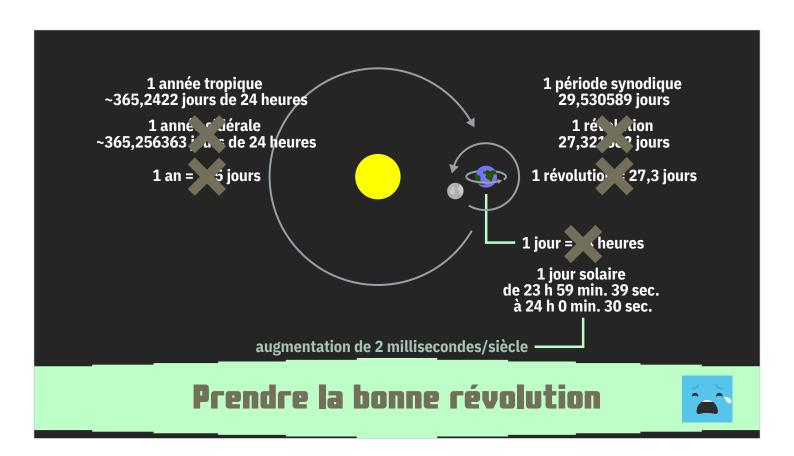
Le mieux est de les éviter mais on ne peut pas toujours : voici les dates.



On considère souvent que la terre met une année de 365 jours pour faire le tour du Soleil, qu'une journée dure 24 heures, que la Lune met 27,3 jours à tourner autour de la terre.



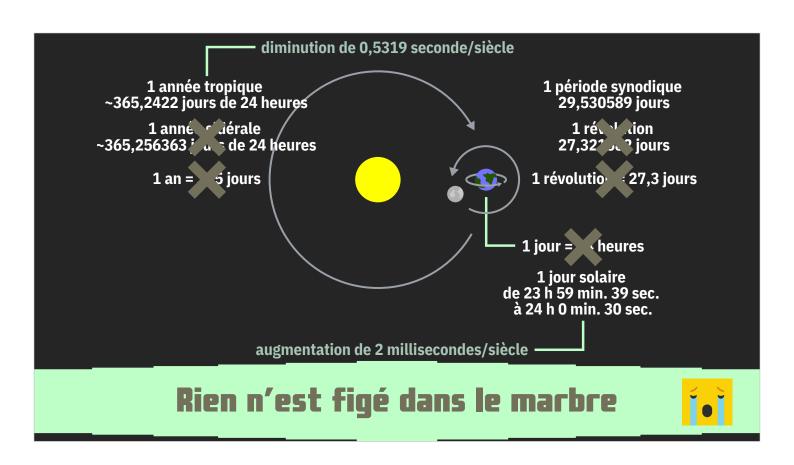
C'est une vision très simpliste. La Terre met en réalité 365,256363 jours de 24 heures à faire un tour du Soleil. Une journée sur terre dure entre 23 heures 59 minutes et 39 secondes et 24 heures et 30 secondes. Et la Lune met 27,321582 jours de 24 heures à tourner autour de la Terre.



Bon, en réalité, le calendrier grégorien (celui qu'on utilise tous les jours) ne se base pas sur l'année sidérale mais sur l'année tropique ou la durée entre deux solstices d'été qui est légèrement plus courte que l'année sidérale : 365,2422 jours de 24 heures.

Les dates basées sur la lune utilise la période synodique ou la durée entre deux phases identiques de la lune qui est plus longue avec 29,530589 jours de 24 heures.

Et pour simplifier les choses, le jour solaire augmente de 2 millisecondes par siècles.



L'année tropique, quant à elle, diminue d'une demi-seconde par siècle.

Dans 4 millions d'années, il n'y aura plus d'année bissextile.



Simplification encore... l'horloge atomique pose quelques problèmes.

Temps universel coordonné (UTC)

- Temps
 - Temps universel solaire : UT1
 - Temps atomique international : TAI
 - Temps universel coordonné: UTC
- Synchronisation et rattrapage
 - 1958: synchronisation UT1 et TAI
 - 1972 : 10 secondes de décalage, création d'UTC



Après la seconde guerre mondiale, l'ère du nucléaire nous a apporté les horloges atomiques et le temps atomique international (TAI), les horloges les plus précises dont nous disposions actuellement.

Elles sont tellement précises qu'il a fallu inventé une méthode pour synchroniser le TAI avec le temps universel solaire (UT1) utilisé jusqu'alors : le temps universel coordonné ou UTC.

Il a été mis en place à partir de 1958. En 1972, un mécanisme de correction est introduit car le temps universel est tout sauf régulier!



- Permet de synchroniser UTC et TAI
- Ajoute ou retranche 1 seconde le 30 juin ou 31 décembre
- · Non prédictible!
- 23 h 59 m 60 s ≠ 24 h 00 m 00 s



La seconde intercalaire a été introduite en 1972 pour corriger le problème.

Elle consiste à ajouter ou retrancher 1 seconde le 30 juin ou 31 décembre pour se resynchroniser l'horloge atomique.

Contrairement à la plupart des mécanismes jusqu'alors utilisés, il n'est pas possible de déterminer à l'avance l'introduction des prochaines secondes intercalaires.

La seule façon d'avoir une horloge système à jour et d'avoir des calculs prenant en compte la seconde intercalaire consiste à mettre à jour régulièrement son système.

La seconde intercalaire est ajoutée à la dernière heure. On obtient donc l'heure 23h59m60s.

Histoire de bugs

- Calendrier Julien vs Grégorien
 Problème de calcul des années bissextiles amenant à la suppression de 11 jours
- Année zéro vs an 1 21° siècle débutant en 2001, naissance de Jésus estimée entre -7 et -5
- Bug de l'an 2000 et années codées sur 2 chiffres Problème soulevé dès 1958 par Bob Berner, co-inventeur du code ASCII
- Bug de l'an 2038 et systèmes 32 bits Fichiers ZIP, système de fichiers FAT, systèmes d'exploitation, horloges temps réel...
- Et tant d'autres...



La seconde intercalaire est loin d'être le premier problème soulevé par le temps.

La calendrier julien a dû être transformé en calendrier grégorien pour corriger le calcul des années bissextiles, supprimant 11 jours au passage au 16^e siècle pour corriger l'erreur.

Le calendrier grégorien ne connaît pas l'année 0. Et il se basait sur une date de naissance de Jésus erronée.

Les limites des premiers systèmes informatiques amenaient les programmeurs à coder les années sur 2 chiffres, entraînant le bug de l'an 2000.

Les premiers timecode Unix sont basés sur des nombres de 32 bits, entraînant une limite à l'année 2038.

Et tant d'autres...

Fuseaux horaires, heure d'été...

- Le fuseau horaire et l'heure d'été dépendent
 - du pays
 - de la date (raison historique)
- Le décalage horaire peut être de 30 ou 45 minutes!



Les fuseaux horaires et l'heure d'été mettent également leur petite pagaille.

Elles dépendent du pays et de la date. Certains pays changent de fuseau horaire en conséquence de conflit ou de négociation, décident d'appliquer ou d'arrêter l'usage de l'heure d'été.

À noter : un décalage horaire n'est pas figé sur un nombre d'heures pleines, il peut être de 30 ou 45 minutes.

Précision du type de données

- Langage de programmation
 - Java 8: 1 ns
 - Python: 1 µs
 - JavaScript
 - API Date: 1 ms
 - API Performance: 5 µs
- Norme
 - **ISO 8601**:1 µs?

- Système de fichiers
 - **FAT**: 2s
 - **EXT3**: 1s
 - **EXT4**: 1 ms
 - **NTFS**: 0,1 μs



Si le comportement des entiers et des flottants reste proche entre les différents langages de programmation, il n'en va pas de même pour les dates.

C'est chacun pour sa pomme!

Les dates Java 8 ont une précision de l'ordre de la nanoseconde, Python de la microseconde et JavaScript de la millisecondes (ou 5 microsecondes avec l'API Performance).

L'encodage des dates dans les systèmes varie elle aussi grandement. Le système de fichier FAT a une précision de 2 secondes, EXT3 (Linux) de la seconde, EXT4 de la milliseconde, NTFS de 100 nanosecondes etc.

Et avec ISO 8601 ? Autour de la microseconde mais cela n'est pas spécifié dans la norme !

Précision réelle

- Atténuation d'attaques Meltdown/Spectre
- Atténuation de prise d'empreinte
- Instabilité de l'horloge système
 - précision du quartz
 - variations en fonction de la température
 - coordination NTP
- Comportement dans une VM suspendue?



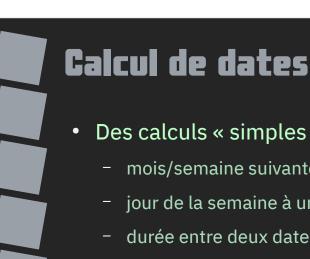
Les API utilisée pour récupérer la date en cours souffrent également de problèmes de précision.

Suite à la découverte des attaques Metldown et Spectre un « flou » de précision a été programmé dans les API des navigateurs afin d'atténuer la possibilité de ces attaques.

Firefox dispose aussi de mécanismes limitant la prise d'empreinte basée sur les timings.

Côté matériel, l'horloge de votre système fonctionne relativement au quartz utilisé par votre processeur. Quartz dont la précision dépend de sa température de fonctionnement. Même en utilisant des protocoles comme NTP, une dérive se produit toujours.

Que dire encore du comportement de l'horloge dans une machine virtuelle qu'on suspend ?



- Des calculs « simples »
 - mois/semaine suivante
 - jour de la semaine à une date
 - durée entre deux dates
- Des calculs moins simples
 - dates religieuses basées sur la Lune...

- Des notions locales
 - numéro de la semaine
 - premier jour de la semaine
 - jours ouvrés



Les dates sont des types complexes pour lesquels un énoncé simple entraîne souvent des questions complexes et des cas aux limites non soupçonnés.

La notion de numéro de la semaine change d'un pays à l'autre comme la notion du premier jour de la semaine (samedi, dimanche ou lundi?).

Et il y a également des calculs moins simples comme les dates religieuses basées sur les cycles de la lune...



Pâques est le dimanche qui suit le 14° jour de la Lune qui atteint cet âge le 21 mars ou immédiatement après

CONCILE DE NICÉE, 325



Jusqu'au Concile de Nicée en 325, de nombreuses dates de Pâques étaient retenues.

Selon le Concile de Nicée en 325, « Pâques est le dimanche qui suit le 14° jour de la Lune qui atteint cet âge le 21 mars ou immédiatement après. »

ISO 8601:2004

- Particularités
 - calendrier grégorien proleptique
 - avec une année zéro
 - accord nécessaire pour les années < 1582
 - support de la seconde intercalaire
- Basé sur des chaînes de caractères
- Nécessite une conversion pour l'affichage local



Il existe une norme ISO destinée à lever les ambiguïtés de lecture et faciliter les échanges d'information. C'est la norme ISO 8601.

Elle étend le calendrier Grégorien sur les années qui le précède et possède une année zéro. Pour toutes les années antérieures à 1582, la norme ne définit pas de comportement particulier. Il est alors nécessaire de s'accorder entre l'émetteur et le récepteur.

Elle est basée sur sur des chaînes de caractères et offre une certaine variété d'écriture.

Il est préférable de gérer cette variété (et ses cas particuliers) à l'aide d'une bibliothèque ou de fonctions dédiées.

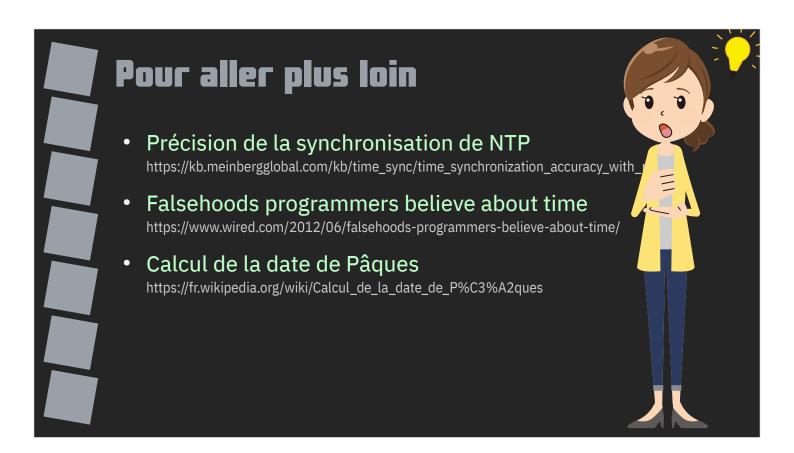


Alors, que faire?

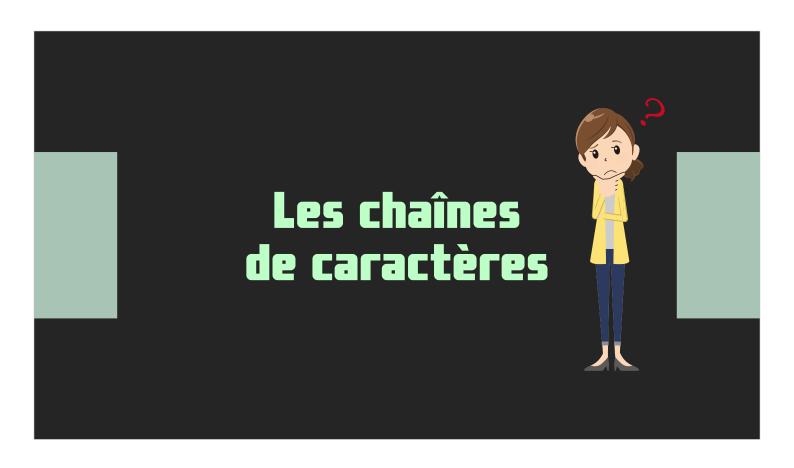
Ne sous-estimez pas les problématiques, faites appel aux normes et standards, utilisez des bibliothèques dédiées.

Mettez à jour régulièrement. La seconde intercalaire n'est pas la seule variable : un pays peut changer à tout moment sa pratique de l'heure d'été ou changer de fuseau horaire.

Multiplier, encore et encore, les tests. Testez, testez, testez!



Voici quelques liens pour aller plus loin dans sa compréhension des dates.



La chaîne de caractères est le type de données le plus versatile.

Cette versatilité a un coût et une dette technique souvent négligés.



Une chaîne de caractères est à la fois conceptuellement une suite ordonnée de caractères et physiquement une suite ordonnée d'unités de code

CHAÎNE DE CARACTÈRES, WIKIPÉDIA



Selon Wikipédia, « Une chaîne de caractères est à la fois conceptuellement une suite ordonnée de caractères et physiquement une suite ordonnée d'unités de code. »

Cette définition est une mise en garde : il va y avoir des concepts...



- **15**e siècle : typographie Caractères, signes, ligatures, espaces
- 16° siècle : apparition des accents en français
- 19° siècle : télégraphe
 Premières codifications et automatisations, codes de contrôle
- 1930 : téléscripteur Codes de contrôle, séparateurs, augmentation du nombre de caractères
- Depuis le 19^e siècle : standardisation Code Baudot, Western Union, EBCDIC, ASCII, ISO-8859-*...



La chaîne de caractères fait appel à des concepts nés bien avant l'informatique.

On peut remonter jusqu'au 15° siècle et l'invention de la typographie par Gutenberg.

Au 16^e siècle apparaissent les accents en français pour préciser des consonances que les lettres latines ne permettaient pas de distinguer.

La révolution industrielle et l'arrivée du télégraphe sonnent l'arrivée des premiers encodages de caractères, encodages qui vont se développer ensuite pour les téléscripteurs et qui seront adaptés à l'outil informatique.

La recherche de standardisation a commencé dès le 19e siècle.

Caractère est un concept

- Représentation visuelle
- Un caractère peut représenter
 - lettre, ligature, sinogramme, emoji, symbole...
 - diacritique (suscrit, souscrit, prescrit, adscrit, inscrit, circonscrit)
 - signe de ponctuation
 - séparateur (espace, tabulation, retour à la ligne...)
 - opération spéciale (sonnerie, effacement, déplacement...)



« Caractère » est un terme qui regroupe plusieurs concepts, rendant sa définition difficile et son emploi trompeur.

Un caractère peut représenter une ou plusieurs lettres, un accent, un signe de ponctuation, un séparateur ou une opération spéciale.

La notion de caractère n'est pas suffisante pour aborder les notions de composition et de formatage. On préfère utiliser le terme point de code.



Pour clarifier la situation, on va parler de glyphe, de point de code et d'encodage.

Le glyphe est la représentation visuelle du caractère, elle est définie par la police de caractères.

Le point de code est un identifiant numérique du caractère. Plusieurs points de code peuvent être nécessaires pour définir une lettre.

L'encodage est la représentation physique du point de code. Les plus connus sont ASCII, UTF-8 et la famille ISO/CEI 8859.

Par exemple, en Unicode, la lettre latine A majuscule est représentée par le point de code U+0041.

Encodages reconnus par l'IANA



US-ASCII ISO_8859-1:1987 ISO_8859-2:1987 ISO_8859-3:1988 ISO_8859-4:1988 ISO_8859-5:1988 ISO_8859-6:1987 ISO_8859-7:1987 ISO_8 8:1988 ISO_8859-9:1989 ISO-8859-10 ISO_6937-2-add JIS_X0201 JIS_Encoding Shift_JIS Extended_UNIX_Code_Packed_Format_for_Japanese Extended_UNIX_Code_Fixed_Width_for_Japanese BS_4730 SEN_850200_C IT ES DIN_66003 NS_4551-1 NF_Z_62-010 ISO-10646-UTF-1 ISO_646.basic:1983 INVARIANT ISO_646.irv:1983 NATS-SEFI NATS-SEFI-ADD NATS-DANO NATS-DANO-ADD SEN_850200_B KS_C_5601-1987 ISO-2022-KR EUC-KR ISO-2022-JP ISO-2022-JP-2 JIS_C6220-1969-jp JIS_C6220-1969-ro PT greek7-old latin-greek NF_Z_62-010_(1973) Latin-greek-1 ISO_5427 JIS_C6226-1978 BS_viewdata INIS INIS-8 INIS-cyrillic ISO_5427:1981 ISO_5428:1980 GB_1988-80 GB_2312-80 NS_4551-2 videotex-suppl PT2 ES2 MSZ_7795.3 JIS_C6226-1983 greek7 ASMO_449 iso-ir-90 JIS_C6229-1984-a JIS_C6229-1984-b JIS_C6229-1984-b-add <u>JIS_C6229-1984-hand JIS_C6229-</u> 1984-hand-add JIS_C6229-1984-kana ISO_2033-1983 ANSI_X3.110-1983 T.61-7bit T.61-8bit ECMA-cyrillic CSA_Z243.4-1985-1 CSA_Z243.4-1985-2 CSA_Z243.4-1985-gr ISO_8859-6-E ISO_8859-6-I T.101-G2 ISO_8859-8-E ISO_8859-8-I CSN_369103 JUS_I.B1.002 IEC_P27-1 JUS_I.B1.003-serb JUS_I.B1.003-mac greek-ccitt NC_NC00-10:81 ISO_6937-2-25 GOST_19768-74 ISO_8859-supp ISO_10367-box latin-lap JIS_X0212-1990 DS_2089 us-dk dk-us KSC5636 UNICODE-1-1-UTF-7 ISO-2022-CN ISO-2022-CN-EXT UTF-8 ISO-8859-13 ISO-8859-14 ISO-8859-15 ISO-8859-16 GBK GB18030 OSD_EBCDIC_DF04_15 OSD_EBCDIC_DF03_IRV OSD_EBCDIC_DF04_1 ISO-11548-1 KZ-1048 ISO-10646-UCS-2 ISO-10646-UCS-4 ISO-10646-UCS-Basic ISO-10646-Unicode-Latin1 ISO-10646-J-1 ISO-Unicode-IBM-1261 ISO-Unicode-IBM-1268 ISO-Unicode-IBM-1276 ISO-Unicode-IBM-1264 ISO-Unicode-IBM-1265 UNICODE-1-1 SCSU UTF-7 UTF-16BE UTF-16LE UTF-16 CESU-8 UTF-32 UTF-32BE UTF-32LE BOCU-1 UTF-7-IMAP ISO-8859-1-Windows-3.0-Latin-1 ISO-8859-1-Windows-3.1-Latin-1 ISO-8859-2-Windows-Latin-2 ISO-8859-9-Windows-Latin-5 hp-roman8 Adobe-Standard-Encoding Ventura-US Ventura-International DEC-MCS IBM850 PC8-Danish-Norwegian IBM862 PC8-Turkish IBM-Symbols IBM-Thai HP-Legal HP-Pi-font HP-Math8 Adobe-Symbol-Encoding HP-DeskTop Ventura-Math Microsoft-Publishing Windows-31J GB2312 Big5 macintosh IBM037 IBM038 IBM273 IBM274 IBM275 IBM277 IBM278 IBM280 IBM281 IBM284 IBM285 IBM290 IBM297 IBM420 IBM423 IBM424 IBM437 IBM500 IBM851 IBM852 IBM855 IBM857 IBM860 IBM861 IBM863 IBM864 IBM865 IBM868 IBM869 IBM870 IBM871 IBM880 IBM891 IBM903 IBM904 IBM905 IBM918 IBM1026 EBCDIC-AT-DE EBCDIC-AT-DE-A EBCDIC-CA-FR EBCDIC-DK-NO EBCDIC-DK-NO-A EBCDIC-F1-SE EBCDIC-F1-SE-A EBCDIC-FR EBCDIC-IT EBCDIC-PT EBCDIC-ES EBCDIC-ES-A EBCDIC-ES-S EBCDIC-UK EBCDIC-US UNKNOWN-8BIT MNEMONIC MNEM VISCII VIQR KOI8-R HZ-GB-2312 IBM866 IBM775 KOI8-U IBM00858 IBM00924 IBM01140 IBM01141 IBM01142 IBM01143 IBM01144 IBM01145 IBM01146 IBM01147 IBM01148 IBM01149 Big5-HKSCS IBM1047 PTCP154 Amiga-1251 KOI7-switched BRF TSCII CP51932 windows-874 windows-1250 windows-1251 windows-1252 windows-1253 windows-1254 windows-1255 windows-1256 windows-1256 windows-1258 TIS-620 CP50220

http://www.iana.org/assignments/character-sets/character-sets.xhtml 2021-01-04

Il y une très grande variété d'encodages. Elle est la conséquence de plusieurs facteurs : les ressources limitées de l'informatique historique (capacité de traitement, de stockage, de bande passante...), le protectionnisme ou encore la propriété intellectuelle.



- Encodages + jeu de caractères
 - **ASCII**: 1963-1986

Encodage 7 bits utilisé par de très nombreuses variantes

- **ISO/CEI 8859**: 1986-2001

Extension de l'ASCII à 8 bits incluant le support international

- **ISO/CEI 10646**: 1993-2012

Encodage d'Unicode: UCS-2, UCS-4, UTF-1, UTF-8, UTF-16, UTF-32

- Caractères uniquement
 - Unicode: 1991-2020

Liste de caractères et d'algorithmes, prise en compte de la polysémie

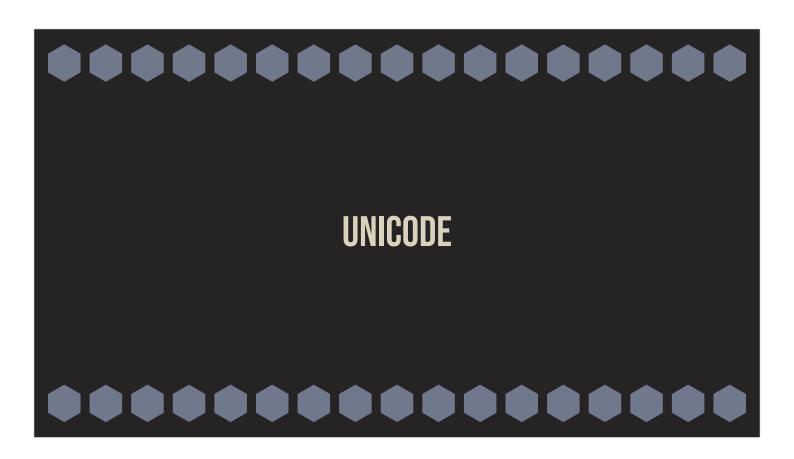


Pendant longtemps, les normes mélangeaient à la fois l'encodage et le jeu de caractères.

Avec le développement des communications et l'arrivée d'internet, le besoin de disposer d'une liste commune de caractères s'est imposée : c'est l'Unicode.

Les encodages les plus utilisés se sont développés autour de l'ASCII et de ses 128 caractères. Le 8^e bit laissé libre par l'ASCII a permis de développer la famille ISO/CEI 8859 ou encore l'UTF-8.

À noter : l'Unicode n'est pas un encodage ! Plusieurs encodages sont disponibles pour l'Unicode.



Faisons donc un petit tour des particularités d'Unicode.

Unicode à la rescousse



- Liste de nombreux caractères
 - 143 859 caractères à la version 13, emojis inclus
 - 1114 111 points de code maximum
 - Unicode ne contient pas tous les caractères possibles!
- Règles d'utilisation
 - Algorithmes de manipulation de chaînes Unicode
 - Bibliothèques ICU4C (C, C++), ICU4J (Java) et ICU4X



Unicode est une liste de caractères et de règles d'utilisation de ces caractères. Unicode ne définit pas l'aspect visuel des caractères.

Cela représente plus de 140000 caractères à la version 13 (2020) et inclus des emojis.

Unicode n'a pas vocation à recenser tous les caractères possibles et imaginables. Par exemple les caractères graphiques que l'on peut retrouver sur les premiers micro-ordinateurs familiaux n'ont pas tous leur équivalent en Unicode.

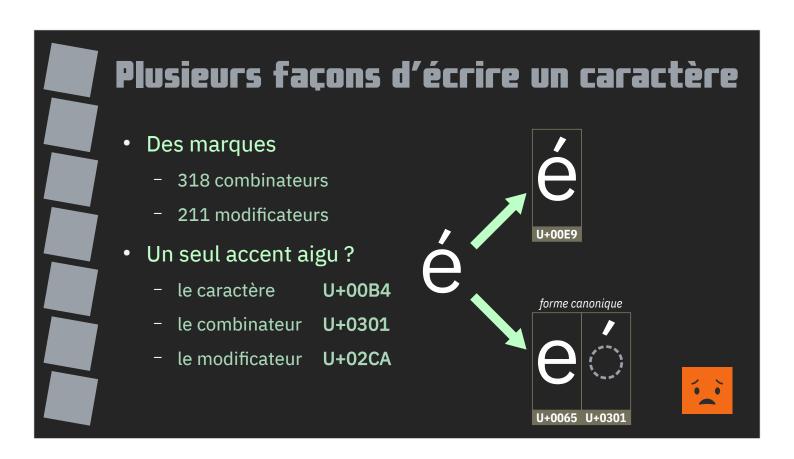
Unicode définit des règles d'utilisation. Une partie de ces règles est implémentée par le projet ICU.



Unicode est un projet conséquent.

Variation_Selector, Vertical_Orientation, White_Space, xdigit

De très nombreuses propriétés sont définis pour chaque caractère référencé par la norme.



Première particularité d'Unicode, il n'existe pas qu'une seule et unique façon d'écrire un caractère.

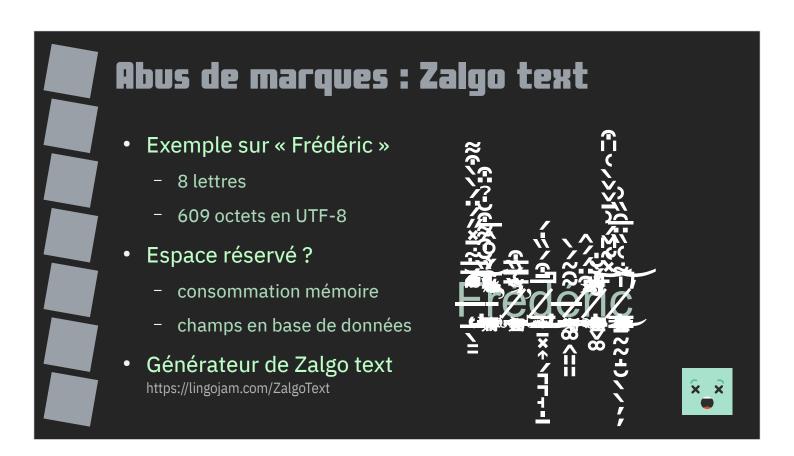
Cela tient notamment du fait qu'Unicode retranscrit beaucoup de systèmes d'écriture et que leur complexité impose cette situation.

Si on ne prend en compte que le français et ses accents, Unicode dispose par exemple du point de code U+00E9 pour désigner le E minuscule accent aigu pour des raisons historiques (c'est comme cela que les encodages historiques encodaient cette lettre).

La forme canonique de cette lettre en Unicode est une lettre E minuscule suivie d'un point de code combinateur accent aigu.

Du point de vue Unicode, ces 2 formes sont équivalentes.

Unicode distingue 318 combinateurs et 211 modificateurs. L'accent aigu dispose lui de 3 points de code : son glyphe, son combinateur et son modificateur.



Unicode ne définit pas de limites quant au nombre de marques qu'on peut ajouter à une lettre.

Cela est exploité par la technique du Zalgo text qui consiste à surcharger chaque lettre d'un mot de marques sans aucune considération linguistique.

En dehors de l'aspect bidouille, cette technique pose des questions techniques : pour un champ pouvant accueillir 8 lettres, quel espace mémoire doit-on réserver afin de pouvoir accueillir un nombre raisonnable de combinateurs ?

À l'assaut de la polysémie



- Unicode distingue les caractères en fonction de leur sens quitte à empiéter sur le domaine des polices de caractères (graisse, empattement, chasse...)
- Certains caractères sont alors confusants (et dangereux) https://util.unicode.org/UnicodeJsps/confusables.jsp

Lettre minuscule latine E e e Minuscule mathématique grasse E de ronde

Lettre minuscule cyrillique ié e e Minuscule mathématique gothique E

Lettre minuscule cyrillique tché abkhaze e e Minuscule mathématique ajourée E

Symbole estimé e e Minuscule mathématique gothique grasse E Minuscule E de ronde e e Minuscule mathématique sans empattement E

Minuscule E italique ajouré e e Minuscule mathématique grasse sans empattement E

Lettre minuscule latine E gothique e e Minuscule mathématique italique sans empattement E

Minuscule mathématique italique E e e Minuscule mathématique à chasse fixe E

Minuscule mathématique grasse E e e Minuscule mathématique italique grasse sans empattement E

Minuscule mathématique italique grasse E $oldsymbol{e}$ e Lettre minuscule latine E pleine chasse

Unicode a cherché à répondre aux problèmes de polysémie introduits par les anciens systèmes.

Chaque caractère doit avoir une signification et utilisation précise quitte a créer plusieurs caractères se ressemblant.

Dans le cas de la lettre latine E minuscule, Unicode définit en tout 20 caractères s'en rapprochant.

Ce problème est bien connu d'Unicode : pour chaque caractère, la base de caractères d'Unicode définit les éventuelles confusions possibles.



Un peu d'espace

• ASCII: 1 espace

SPACE

• ISO/CEI 8859: 2 espaces

SPACE / NO-BREAK-SPACE

• Unicode: 18 espaces

SPACE / NO-BREAK SPACE / OGHAM SPACE MARK / MONGOLIAN VOWEL SEPARATOR / EN QUAD / EM QUAD / EN SPACE / EM SPACE / THREE-PER-EM SPACE / FOUR-PER-EM SPACE / SIX-PER-EM SPACE / FIGURE SPACE / PUNCTUATION SPACE / THIN SPACE / HAIR SPACE / NARROW NO-BREAK SPACE / MEDIUM MATHEMATICAL SPACE / IDEOGRAPHIC SPACE



Cette chasse à la polysémie se traduit jusque dans les espaces.

Unicode n'en définit pas moins de 18, là où l'ASCII n'en avait qu'un.



- « Vraie » ligature
 - égalité si translittération
 - œuvre ≠ oeuvre
- « Fausse » ligature
 - héritée de la typographie
 - égalité en forme NFKD/NFKC
 - effleurer ≃ effleurer
 - ex.: ff, fi, fl, ffi, ffl, ft, st

- Ligature visuelle
 - générée par la police
 - figure = figure



S'il faut parfois plusieurs points de code pour identifier une lettre, Unicode peut aussi n'utiliser qu'un seul point de code pour représenter plusieurs lettres : c'est le cas notamment des ligatures.

Le e dans l'o du français (comme dans œuf, bœuf, œil ou œuvre) est considéré comme une seule et même lettre, différente du o suivi de e (œuf est un mot de 3 lettres et non de 4 en français).

Pour des raisons historiques, Unicode intègre des ligatures typographiques comme ff, fi, ffl, ffl, ffl, ft, st alors que la ligature devrait être réalisée au niveau du rendu visuel, donc en dehors du domaine d'Unicode. Les propriétés Unicode de ces caractères permettent de les transformer en leur équivalent canonique.

Si ces dernières existent, il est cependant préférable de laisser le rendu graphique de la police de caractères s'en charger.



- Parcours de chaînes
- Encodage
- Conversion
- Détection d'encodage
- Compression
- Locales
- Normalisation

- Formatage
- Découpage
- Translittération
- Collation / tri
- Recherche
- Mise en page
- Etc.



Pris indépendamment, il y a déjà beaucoup à dire sur chaque caractère Unicode.

Inclus dans une chaîne de caractères, les choses se corsent!

De nombreuses opérations sont alors possibles et aucune n'est triviale : parcourir une chaîne, l'encoder, la convertir, détecter l'encodage, normaliser les caractères, formater, découper, trier, rechercher, mettre en page etc.

Les règles étant nombreuses, dépendantes de la langue ou du contexte d'utilisation, le projet ICU a été mis en place avec pour but de produire une bibliothèque de fonctions de manipulations de chaînes Unicode conformes aux règles.

Une petite collation?

- Trier des chaînes de caractères est complexe!
 - tous les caractères ne sont pas triables : Ж, ╝, Ӆ, ∞, ◊, △...
 - le point de code n'est pas un indicateur d'ordre!
 - plusieurs niveaux et ordres de comparaison (caractères, accents...)
- Dépend de : destinataire, langue, culture, personnalisation
- Points de code contrôlant la collation
 - Combining Grapheme Joiner **U+034F**
 - Bidirectional Ordering Controls



Prenons l'exemple de la collation (ou tri de chaînes de caractères).

C'est une opération complexe : tous les caractères ne sont pas triables, le point de code n'est pas un indicateur d'ordre et il existe plusieurs niveaux et ordres de comparaison.

La collation dépend du destinataire, de sa langue, de sa culture, de sa personnalisation et pas nécessairement de la langue-même du contenu à trier.

Pour complexifier encore plus, il existe des points de code contrôlant le comportement de la collation.



Conventions linguistiques complexes

- De langue
 - **suédois** z < ö
 - allemand ö < z
- D'usage
 - dictionnaire allemand of < öf
 - annuaire allemand öf < of

- De personnalisation
 - minuscules *vs* majuscules
 - ordre des accents
 - ex. cote, côte, coté, côté
- Des ligatures



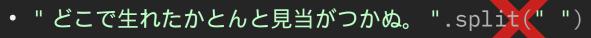
Pour illustrer la situation, on peut prendre le cas du suédois et de l'allemand qui ne classent pas le o tréma et le z de la même façon.

Le dictionnaire allemand ne traite pas le 0 et le 0 tréma de la même façon que l'annuaire allemand.

La collation peut également être personnalisée pour préciser le comportement vis-à-vis des minuscules et des majuscules ou l'ordre des accents. Comment classeriez-vous cote, côte, coté, côté?

Il faut encore prendre en compte le cas des ligatures qui va imposer de comparer un nombre de points de code différents (on ne peut pas trier en regarder point de code par point de code).

Découper un texte



- どこ/で/生れた/か/とんと/見当/が/つかぬ
- les langues n'utilisent pas toutes les espaces pour séparer les mots
- traduction : je n'ai aucune idée de l'endroit où il/elle est né/e
- La ponctuation est différente en fonction de la langue



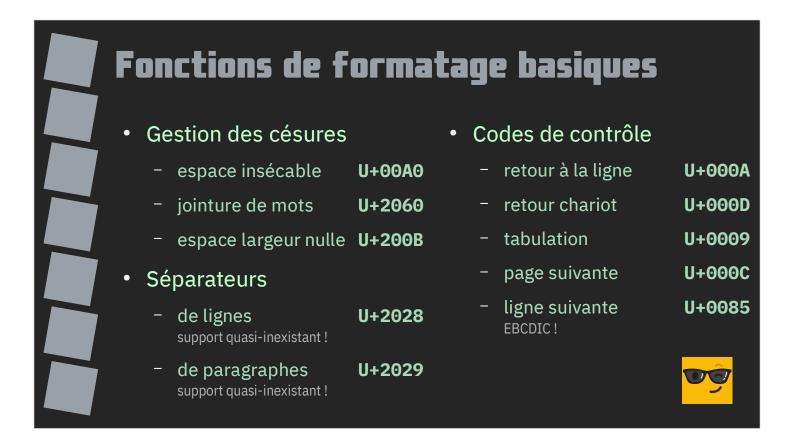
Autre exemple : comment découper un texte en mot ?

Naïvement, un développeur occidental va rechercher des espaces, en oubliant pour commencer qu'il existe plusieurs espaces.

Cette approche ignore complètement que certaines langues n'utilisent aucun espace pour séparer leurs mots comme c'est le cas pour le japonais.

Le cas du japonais nécessite à lui seul un algorithme dédié.

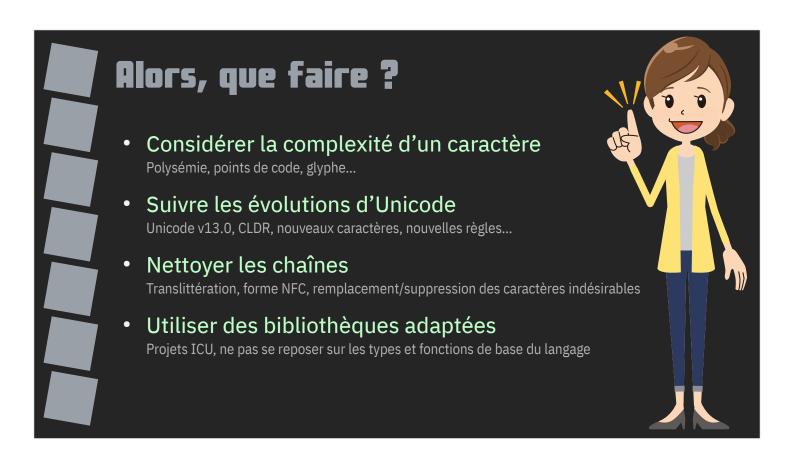
Chasse à la polysémie oblige, la ponctuation est différente en fonction de la langue (même si le rôle est similaire).



Une petite dernière pour la route : les chaînes Unicode peuvent comporter des points de code dédiés au formatage basique.

Cela inclut la gestion des césures (le fait de couper un mot pour le continuer sur la ligne suivante), la séparation des lignes et des paragraphes.

Pour des raisons historiques, Unicode distingue les codes de contrôles retour à la ligne, retour chariot et ligne suivante (ce dernier venant de l'encodage EBCDIC).



Alors, que faire?

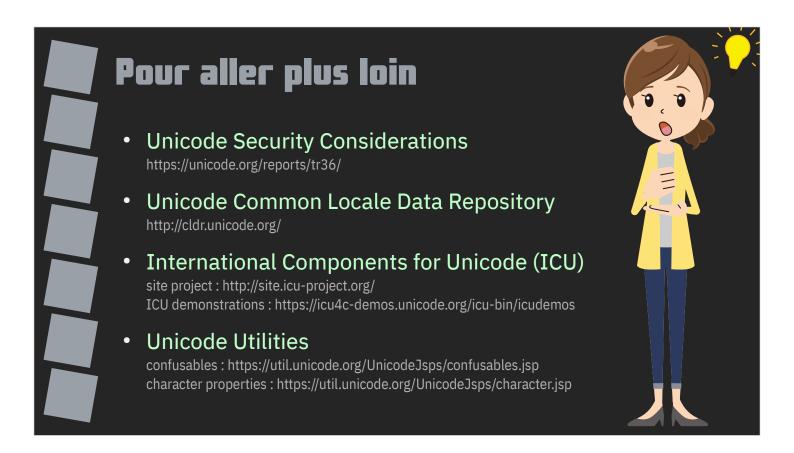
Quand vous devez manipuler une chaîne Unicode, vos sens doivent être en alerte maximum! Nous n'avons fait qu'effleurer la complexité de l'Unicode, la littérature produite par son consortium est énorme.

Et la norme continue d'évoluer.

L'intérêt premier d'Unicode réside dans un socle commun d'échange d'information. C'est à vous de savoir ce que doivent devenir ces données une fois qu'elles intègrent votre système d'informations.

C'est votre responsabilité de nettoyer et valider les chaînes que vous recevez, même si elles proviennent de champs « limités » par ce que l'utilisateur peut saisir.

Utilisez des bibliothèques adaptées qui pourront gérer pour vous tous les cas tordus auxquels vous ne pourrez pas penser.



Pour aller plus loin voici quelques liens sur lesquels jeter un œil.



Nous n'avons fait que parler d'Unicode sans jamais évoquer le concret : comment ça se passe quand on programme ?

Chaînes et caractères

- C → suite d'octets à zéro terminal aucune limite de taille, aucune contrainte hormis l'impossibilité d'utiliser le caractère \000
- Python 3 → tableau de points de code Unicode valeurs limitées à sys.maxunicode
- Haskell → liste chaînée de points de code Unicode valeurs limitées à maxBound :: Char, plusieurs autres types de chaînes de caractères existent
- JavaScript → tableau de mots de 16 bits aucune contrainte
- PHP → tableau d'octets aucune contrainte



Les chaînes de caractères se ressemblent beaucoup d'un langage à l'autre mais leurs différences sont fondamentales et ont des conséquences et contraintes importantes.

Le C, historiquement, utilise une suite d'octets à zéro terminal. Aucune notion réelle d'encodage ou de contrainte hormis l'impossibilité d'utiliser le caractère NUL.

En Python 3, une chaîne est un tableau de points de code Unicode. Les points de code ont des limites, on ne peut donc pas y stocker de valeur arbitraire.

Haskell utilise aussi des points de code mais, en standard, sous forme de liste chaînée.

JavaScript utilise un tableau de mots de 16 bits (issu de l'UCS-2 puis de l'UTF-16).

Quant à PHP, il s'agit d'un tableau d'octets sans encodage ni contrainte.

JavaScript et ses chaînes

- 3 opérations
 - stricte : a === b
 - faible : a == b
 - « très » stricte : Object.is(a, b)
- Comparaison abstraite
 - [1, 2] == "1,2"
 - 10 == "1e1" mais "1e1" != 10

- 4 algorithmes
 - stricte (===)
 - abstraite (==)
 - SameValue
 - SameValueZero (String.includes)



Dans la plupart des langages, les chaînes de caractères sont des types « basiques » : au mieux, ils prennent en compte quelques aspects de l'Unicode. Les opérations n'existent que pour construire d'autres opérations, pas pour traiter directement des chaînes de caractères. C'est au développeur d'apporter un sens aux différents points de code.

Dans une volonté d'apporter plus de confort au développeur, les langages surchargent parfois leurs opérateurs, quitte à entraîner des comportements inattendus.

Aujourd'hui JavaScript dispose de 3 opérations de comparaison associées à 4 algorithmes.

L'opérateur == va chercher à faire une conversion automatique par rapport au premier objet comparé. Cela a, en plus, pour conséquence de retirer sa commutativité à l'opérateur.

Il faut bien sûr préférer l'opérateur de comparaison strict === quand il est disponible, voire Object.is(a, b).

PHP et ses chaînes À la recherche du nombre perdu!

- - "0000000042" == "42" / "1e1" == "10"
 - 10 == "1e1" *et* "1e1" == 10
 - 0 == "a" *(pour PHP < 8)*
 - md5('240610708') == md5('QNKCDZO')
- Attention aux clés!
 - array(42=>24)[**42**] === array(42=>24)[**"42"**]
 - array(42=>24)[**42**] !== array(42=>24)[**"042"**]



PHP regarde toujours si une chaîne ne contient pas un nombre.

Comparer deux chaînes dont la conversion en nombre donne la même valeur aboutira à une égalité.

Cela a pour conséquence qu'il ne faut jamais utiliser l'opérateur == pour comparer deux hash md5 en PHP : il existe des cas pour lesquels cet opérateur considérera l'égalité comme vraie alors que les deux chaînes sont différentes.

Ce comportement est légèrement différent quand il s'agit de clés de tableau associatif ou la recherche de clé est réalisée sur la conversion en chaîne de caractères de la clé.

Les clé 42, 42.0 et "42" seront considérées comme identiques.

MySQL et ses chaînes

À la recherche du nombre perdu

```
- vrai:SELECT 10 = "1e1";
- vrai:SELECT "1e1" = 10;
- vrai:SELECT 0 = "a";
- faux:SELECT "1e1" = "10";
- faux:SELECT md5('240610708') = md5('QNKCDZO');
```

• Pas de comparaison sur la forme canonique Nécessité de normaliser les chaînes Unicode avant leur insertion en base de données



MySQL présente une gestion des comparaisons de chaînes de caractères qui se rapproche de celle de PHP mais avec quelques différentes.

Il est capable par exemple de faire une conversion de type à la volée pour comparer une chaîne de caractères avec un nombre.

Si les bases des données gèrent le stockage des informations en Unicode (avec encodage au choix UTF-8, UTF-16 etc.), toutes les bases de données ne permettent pas de faire des comparaisons en suivant les règles de l'Unicode et se bornent à des comparaisons strictes.

MySQL n'est notamment pas capable de comparer les formes canoniques de deux chaînes Unicode. Pour y parvenir, il faudra réaliser l'opération côté application ou prévoir une normalisation des chaînes avant leur insertion en base de données.

Littéralement

- Chaque langage a sa propre gestion des chaînes littérales
 - plusieurs types de chaînes littérales
 - usage généralisé trompeur de ' et "
- Une grande variété de caractères spéciaux



Chaque langage dispose de ses propres chaînes littérales. En général, elles sont encadrées par des quotes simples ou doubles.

Cette similarité est trompeuse car chaque langage a ses spécificités et ses limitations.



- Une chaîne peut contenir
 - une chaîne qui contient
 - une chaîne qui contient
 - une chaîne etc.



- Caractères spéciaux et séquence d'échappement
 - différences entre langages
 - plusieurs types de chaînes par langage
 - plusieurs types de valeurs littérales par langage



La versatilité de la chaîne de caractères permet de jouer aux poupées russes : une chaîne peut contenir une chaîne qui peut contenir une chaîne qui peut...

Vous utilisez cette propriété très souvent : au niveau le plus bas en incluant une chaîne de caractères littérale dans un programme en PHP ou JavaScript, et de manière plus avancée quand une chaîne de caractères littérales contient par exemple une requête SQL.

Et que dire d'un programme PHP qui va générer des commandes qui seront ensuite exécutées par un shell Unix ?

Pour permettre cela, il est obligatoire de recourir à des caractères d'échappement permettant de distinguer l'utilisation d'une caractère spécial.

```
The great escape

php -r 'system("echo \"\\\\\Hello, World\\\\\!\\\\"");'

8 6 9

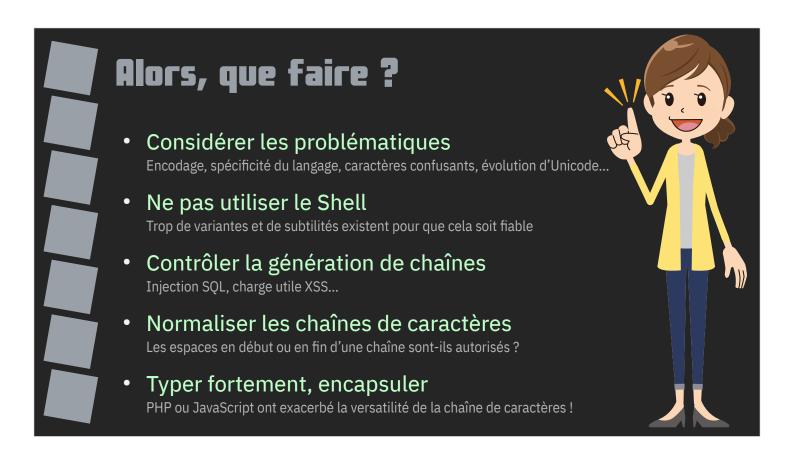
A) Hello, World!

B) \Hello, World!\\
C) \\Hello, World\!\\
D) \\\\Hello, World\\!\\\
```

Exemple simple d'imbrication malsaine : une ligne de commande shell (Unix) qui lance l'interpréteur PHP qui, à son tour, va exécuter une ligne de commande dans le shell. Cette ligne ainsi exécutée va afficher une chaîne.

La bonne réponse est la réponse B.

Si vous avez trouvé la bonne réponse sans hésiter, je vous tire mon chapeau!



Alors, que faire?

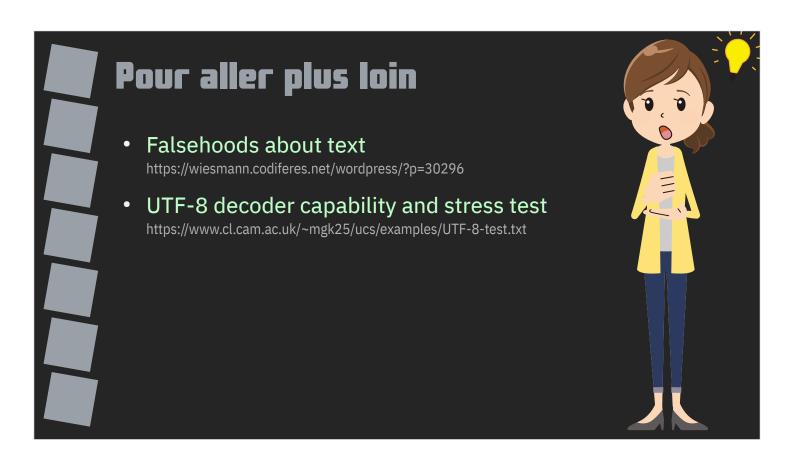
Tout d'abord, considérer que toute chaîne de caractères provenant de l'extérieur de votre fonction n'est pas valide.

Ne jamais utiliser le Shell pour exécuter des programmes depuis votre application, préférer le lancement direct. Il y a beaucoup trop de variantes et de pièges pour en valoir la peine.

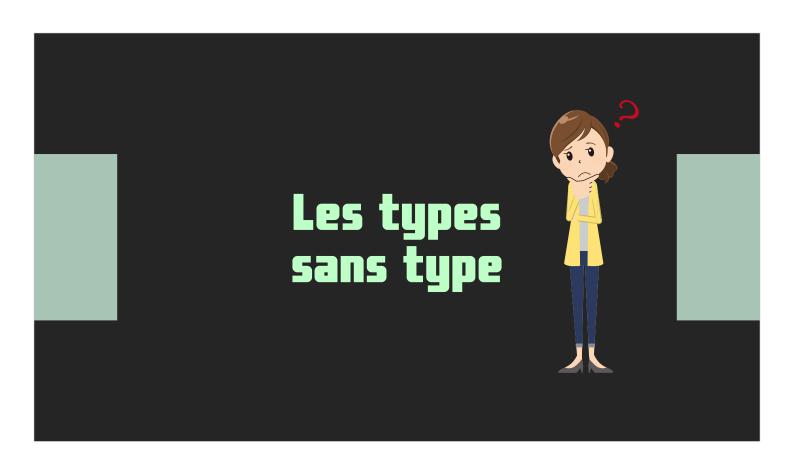
Une chaîne de caractères n'a aucune signification en elle-même, c'est à votre programme de l'apporter : une chaîne de caractères peut représenter un prénom, un nom, une adresse, une URL etc. Y'a-t-il un intérêt à supporter les emojis dans un prénom ? Des codes formatage ?

D'où la sempiternelle recommandation : typer fortement, encapsuler vos données ! Cela limite « by design » l'utilisation d'opérations incohérentes.

Regarder régulièrement ce qu'il se passe autour d'Unicode (apparition de nouveaux codes).



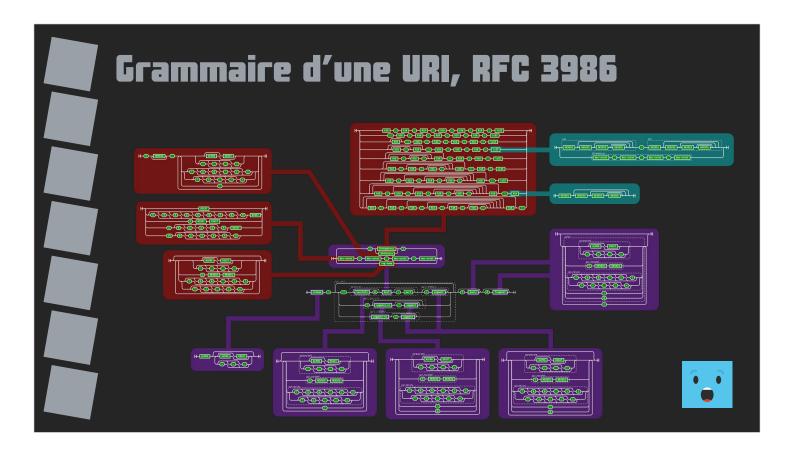
Pour aller plus loin, vous pouvez jeter un œil sur une liste de fausses croyances sur les chaînes de caractères ou sur un stress test de décodeur de chaînes UTF-8.



Un type de données sans type ? Vous en utilisez très souvent dès que vous manipulez une chaîne de caractères sans l'encapsuler. Un exemple ?



Les identifiants de ressources ou URI.



Voici le schéma chemin de fer de la grammaire d'une URI telle que définie par la RFC 3986.

Il montre d'emblée qu'une URI n'est pas une structure basique.

Pour info, toujours selon la RFC 3986, les URL font partie des URI.

IRI, IDN, URI, URL, URN?

• Les "U"

- URI: Unified Resource Identifier

- URL: Unified Resource Location

- URN: Unified Resource Name

• Les "I"

- IRI: Internationalized Resourced Identifier

- IDN: Internationalized Domain Name



Les RFC de l'IETF ont cherché à définir précisément les choses.

À commencer par les URI/URL/URN.

Avec l'adoption internationale massive du web, le besoin de s'ouvrir à d'autres langues a vu l'introduction de IRI/IDN.

Une regex pour IRI, RFC 3987



C'est le sujet de la RFC 3987.

Voici un exemple d'expression rationnelle permettant d'analyser, valider et découper une IRI selon la RFC 3987.

Pas évident à maintenir ou à déboguer...

Un type sans type

- Représenté par / manipulé avec des chaînes de caractères
- Mélange des genres
 - URI → RFC 3986
 - IRI → RFC 3987
 - Systèmes de fichiers
 - Microsoft (MS-DOS, Windows, FAT, NTFS...)
 - Apple (MacOS, MacOSX, HFS...)
 - Les autres (Unix...)



La plupart des URI sont représentées/manipulées avec des chaînes de caractères sans encapsulation, sans réelle considération pour leur complexité ou leur spécificité.

Elles sont mêmes souvent mélangées à des chaînes de caractères provenant de chemins dans des systèmes de fichiers ayant eux-mêmes leurs propres spécificités.

C'est une source potentielle de faille de sécurité.

Un type bien normé ?

- Qui fait les URLs?
 - IETF
 - WHATWG
 - Unicode
 - W3C

- Le monde réel
 - navigateurs
 - robots (indexation...)
 - applications
 - sites web
 - etc.

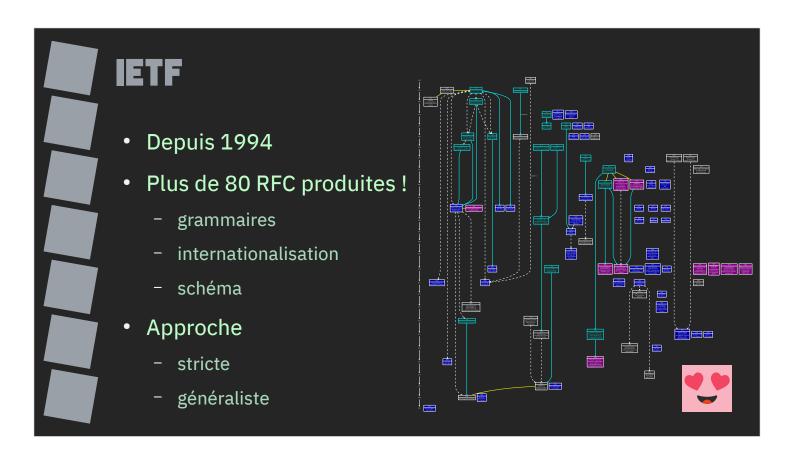


On pourrait penser que les URI soient parfaitement normées.

C'est loin d'être le cas!

Il n'est pas évident de s'y retrouver entre les spécifications produites par les différents groupes et consortium.

Des spécifications quelque peu malmenées lors de leur utilisation dans le monde réel, que ce soit dans les navigateurs, les robots d'indexation, les applications ou les sites web.



L'IETF produit des RFC concernant les URL depuis 1994 avec une approche stricte (définition de grammaires précises) mais généraliste (l'IETF a Internet comme champ d'action, pas uniquement le web).



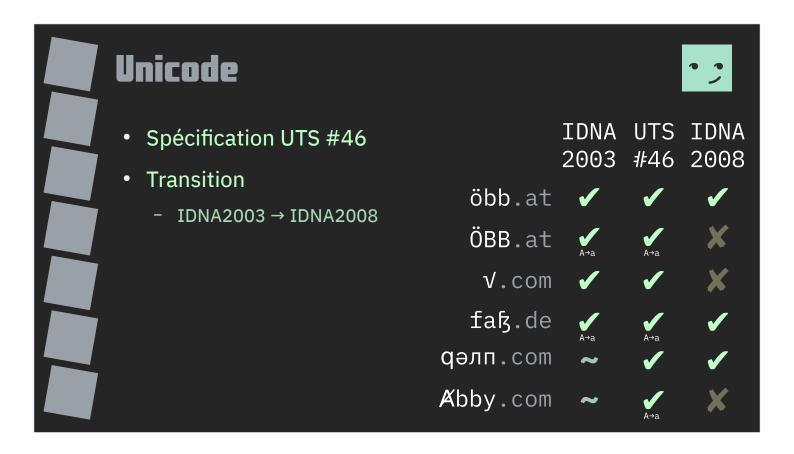
En 2004, Apple, Google, Mozilla et Microsoft crée le WHAT Working Group.

Le but de ce groupe de travail est de créer des standards vivants, à commencer par HTML.

Leurs travaux se sont également portés sur les URL.

Par approche vivante, il faut entendre que le standard évolue continuellement, il n'y a pas de version figée comme l'IETF ou le W3C pourraient produire.

L'approche est également orientée web.



Que vient faire l'Unicode dans cette histoire d'URL?

Le consortium a proposé des spécifications permettant de faciliter la transition d'IDNA2003 à IDNA2008.

IDNA2003 et IDNA2008 sont des spécifications permettant l'utilisation de caractères non-ASCII dans les noms de domaines.

La révision de 2008 n'est pas totalement compatible avec la version de 2003 et pose ainsi des problèmes.

W3C

- WebPlatform.org
 - Adobe, Apple, Facebook, Google, HP, Microsoft, Mozilla, Nokia, Opera, W3C
 - lancé en 2012, arrêté en 2015
- XHTML 2 vs HTML 5
- Recopie à peu près WHATWG...



Le W3C pourrait avoir son mot à dire sur la question mais a mal vécu l'abandon de XHTML 2 au profit de HTML 5.

Il recopie des morceaux de spécifications provenant des travaux de WHATWG malgré la tentative WebPlatform entre 2012 et 2015.

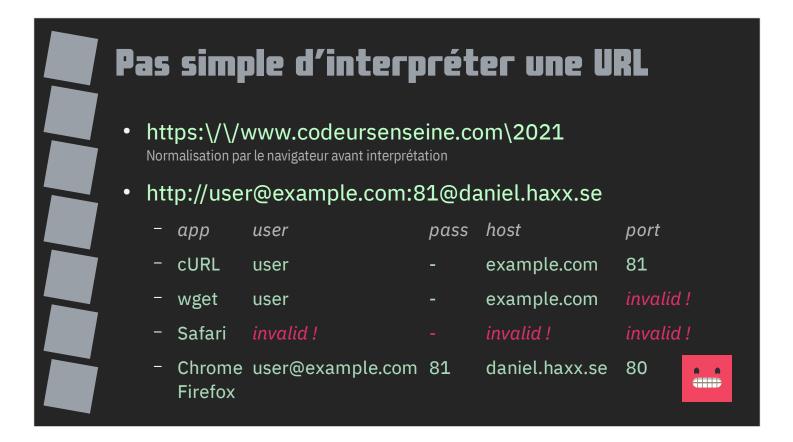


Le W3C copie-colle parfois notre travail sur son propre site web, y appose son propre logo, change le nom des rédacteurs, etc.

> DOMENIC DENICOLA - GOOGLE, ÉDITEUR WHATWG 10/02/2017 - REDDIT



En 2017, Domenic Denicola de Google répondait sur Reddit « Le W3C copie-colle parfois notre travail sur son propre site web, y appose sont propre logo, change le nom des rédacteurs, etc. ».



L'interprétation d'une URL n'est actuellement pas une mince affaire!

Pour « simplifier la vie » des utilisateurs, les navigateurs ont une politique peu rigoureuse quant à l'écriture des URLs, par exemple avec les antislashs sous Chrome et Firefox.

Daniel Stenberg, créateur de cURL, milite d'ailleurs pour une meilleure standardisation et donne l'exemple de l'interprétation d'une URL présentant deux arobases (ce qui va à l'encontre de la RFC 3986). Les comportements varient grandement entre applications et bibliothèques.

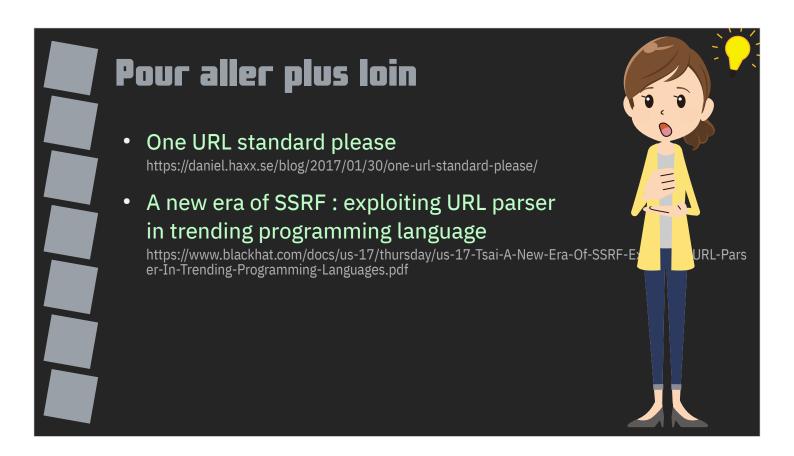


Alors, que faire?

Il n'y a pas que les navigateurs dans la vie ! Un grand nombre de programmes utilisent des URLs pour accéder à des ressources sur internet : des robots d'indexations, des applications mobiles etc.

La génération ou l'inclusion d'URL par vos programmes doit donc faire l'objet d'un soin tout particulier afin d'éviter l'ouverture de faille XSS ou SSRF.

Cela peut notamment passer par une normalisation des URLs que vous traitez, une solution étant d'adopter un sous-ensemble strict d'URL basé, par exemple sur la RFC 3986 ou 3987.



Pour aller plus loin, vous pouvez jeter un œil sur deux liens.

Le premier est le billet de blog de Daniel Stenberg où il argumente le besoin d'une meilleure standardisation des URLs.

Le deuxième est une présentation techniques des failles SSRF exploitant l'analyse des URLs par différents programmes et bibliothèques.



Informatique:

Alliance d'une science inexacte et d'une activité humaine faillible

LUC FAYARD DICTIONNAIRE IMPERTINENT DES BRANCHÉS

Citation de Luc Fayard dans son Dictionnaire impertinent des branchés.

Informatique : alliance d'une sicence inexacte et d'une activité humaine faillible.



Merci de votre attention!

Merci à l'équipe de Codeurs en Seine.

Vous pouvez me retrouver sur les internets aux endroits suivants.