SCI6373 Programmation documentaire

Cours 2

Été 2025

Plan (1/2)

- "Défi" d'hier… *(Breaking news: j'ai oublié d'en parler…)*
- Exercices après C1 - retour
- Arbres d'exécution
- Éléments `<script>` dans une page HTML
- Nouveaux opérateurs
  - Préparation de la lecture des sections 9 et 10

Plan (2/2)

- Interactions de base
- Conversions automatiques de type
- Votre deuxième script
- Pensée algorithmique
  - Patrons de conception (*Design Patterns)*
- Commentaires JS
- Notes sur les constantes
- Varia syntaxique

Arbres d’exécution d'une expression

- Outil graphique
  - Pour mettre en évidence l’ordre d’exécution des différentes opérations dans une expression
  - Démontre qu'on comprend comment une expression est exécuté
  - Structuré en colonnes

Arbre d'exécution (1/3)

- Première colonne :
  - Parcourir l'expression de gauche à droite et empiler l'un sous l'autre les **constantes** et les **noms de variable** rencontrés
  - Opérateurs et parenthèses ignorés, mais…
    - On identifie les opérateurs : ils seront traités dans les colonnes 2 et suivantes

Arbre d'exécution (2/3)

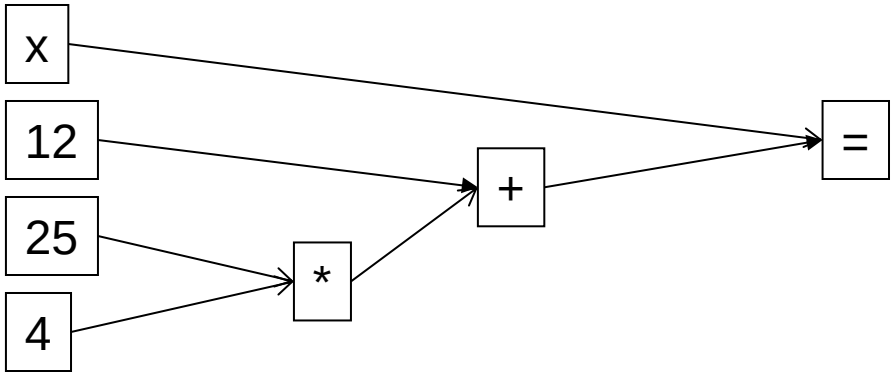
- Colonnes 2 et suivantes, de gauche à droite :
  - Un opérateur par colonne, selon l'ordre d’exécution
  - Chaque opérateur est lié par une flèche à ses arguments dans les colonnes précédentes

Arbre d'exécution (3/3)

– Argument de gauche : la flèche du haut
– Argument de droite : la flèche du bas

Exemple 1

x = 12 + 25 * 4

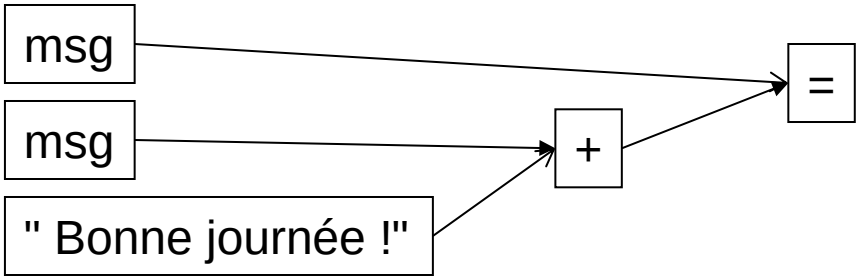


N.B.: On peut connaître le résultat,
car toutes les valeurs sont connues

Résultat : 112

Exemple 2

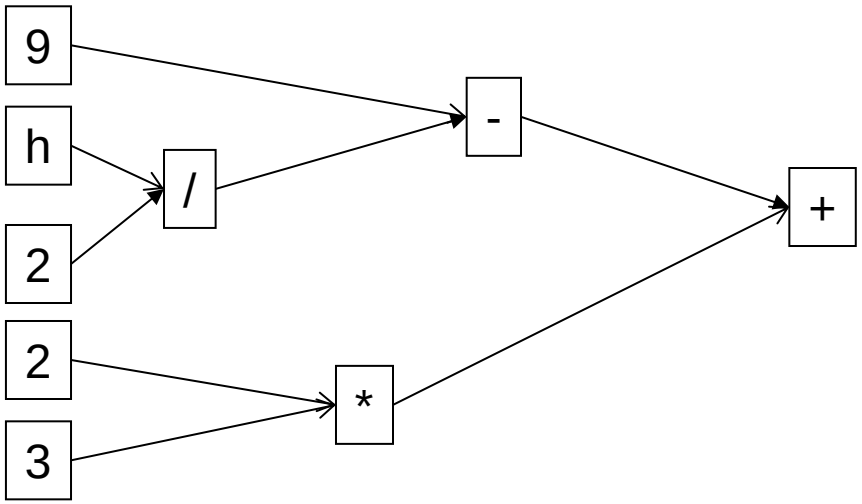
msg = msg + " Bonne journée !"



N.B.: On ne peut pas connaître le résultat, car il dépend de la valeur de `msg`

Exemple 3

9 - (h / 2) + (2 * 3)



N.B.: On ne peut pas connaître le résultat, car il dépend de la valeur de `h`

Conversions automatiques (1/3)

- Lors d'une concaténation avec une valeur caractère, tout autre type de valeur est automatiquement "converti" en chaîne de caractères :

  25 + "a$*" → "25a$*"

  8080 + "" → "8080"

  "abc" + true → "abctrue"

  NaN + "aNa" → "NaNaNa"

  "2" + (5-10) → "2-5"

Conversions automatiques (2/3)

- À l'inverse, lors d'une opération *arithmétique*, une valeur caractère est automatiquement "convertie" en valeur numérique *à condition qu'elle représente un nombre* (sinon → NaN) :

  25 - "2" → 23

  808 * "10" → 8080

  "25.45" / 10 → 2.545

  "abc" - 0 → NaN

Opérations mixtes nombre-caractères (1/2)

20 - 10	10
'20' - 10	10
20 - '10'	10
20 + 10	30
'20' + '10'	'2010'
'20' + 10	*'2010' ou 30 ??*

Opérations mixtes nombre-caractères (1/2)

20 - 10	10
'20' - 10	10
20 - '10'	10
20 + 10	30
'20' + '10'	'2010'
'20' + 10	'2010'

La concaténation l'emporte sur l'addition !

Opérations mixtes nombre-caractères (2/2)

'20' * 10	200
'20' / 10	2
20 * '10'	200
20 / '10'	2
'20' * '1'	20
'20' / '1'	20

Nouveaux opérateurs

On garnit son coffre à outils…

mais dans le respect du principe PYM !

Interactions de base

Interactions de base avec l'utilisatrice (1/2)

`alert("Message")`

- Pour afficher quelque chose à l'utilisatrice
- Ex.: 010-hello-world

- Par la règle d'enchaînement, l'argument peut être *n'importe quelle expression*
  - C'est le résultat de l'expression qui est affiché
- *En tant qu'expression*, alert() retourne comme résultat : `undefined`

Interactions de base avec l'utilisatrice (2/2)

`prompt("Message")`

- Affiche le message + *recueille une réponse de l'utilisatrice*
- Le résultat de prompt en tant qu'expression est la *valeur caractère saisie par l'utilisatrice*
  - L'utilisatrice n'inscrit pas de guillemets !
- Un seul argument, mais on dit "1er argument" !
- Si "Annuler" → `null`
- Si rien saisi → retourne `""` (la chaîne vide)

Votre deuxième script

- Une page scriptée qui demande le nom de l'utilisateur et affiche le message :

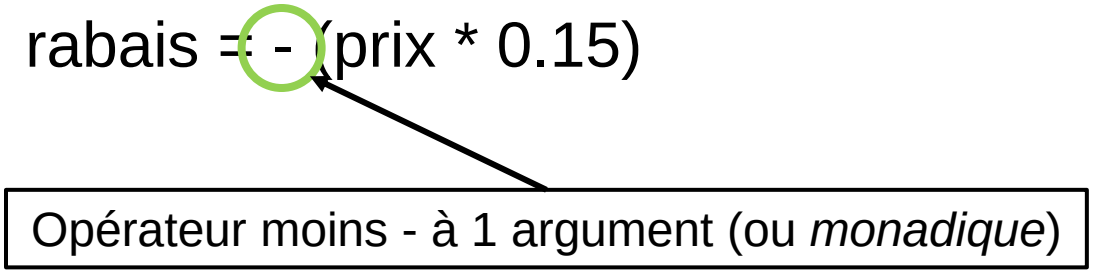
Bonjour XXX

- où "XXX" est le nom fourni par l'utilisatrice
- **Ignorer** les cas "problèmes" possibles :
  - L'utilisatrice entre un nom vide
  - Elle clique sur "Annuler"…

Commencez avec 005 : infrastructure vide
Voir solutions 022 024 030

Les opérateurs - et + *unaires*

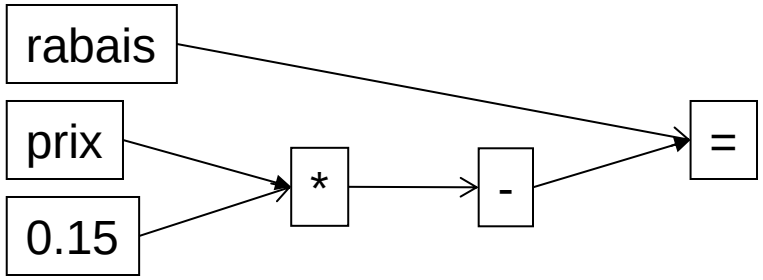
- Le moins (-) et le plus (+) peuvent travailler sur un seul argument, inscrit à leur droite :



- Le - inverse le signe du nombre
- Le + convertit une chaîne en nombre (ou NaN si non numérique)

Exemple 4

rabais = - (prix * 0.15)



N.B.: On ne peut pas connaître le résultat, car la valeur de la variable prix nous est inconnue

Comparaisons

- Donnent comme résultat une valeur "booléenne" (`true` ou `false`)

  ==

  >

  >=

  <

  <=

  !=

Comparaisons mixtes (1/2)

- Comparaison nombre-caractère
  - Une conversion de la chaîne en *valeur numérique* est automatiquement opérée:

```
'0237.00' == 237          true
" 237.1 " > 237           true
```

  - Si la chaîne ne représente pas un nombre, elle est convertie en NaN (*not a number*)

```
42 > "a"                  false
42 <= "a"                 false
42 > NaN                  false
42 <= NaN                 false
```

Comparaisons mixtes (2/2)

- *Attention, donc:*

  '12' > '2' `false`

  '12' > 2 `true`

Comportements étonnants

- À regarder et à… oublier (PYM) !

```
0 == ""             true
0 == "    "         true
80 + null           80
null + null         0
null == 0           false
undefined == 0      false
undefined != 0      true
true + true         2
true == 1           true
NaN == NaN          false
NaN != NaN          true
```

Égalité stricte (===)

- L'opérateur '===' signifie "strictement égal à" : même type de donnée *et* même valeur

  Empêche les conversions automatiques

  !== est sa négation : non strictement égal

  Ex.: 1 === '1' false

  1 !== '1' true

  1 == '1' true

Choix conditionnel ? : (1/2)

- Opérateur à trois opérandes :

  valeur1 **?** *valeur2* **:** *valeur3*
- *valeur1* doit être une valeur booléenne (true ou false)
- *valeur2* et *valeur3* sont des valeurs JS quelconques
- Les trois peuvent bien sûr être une **sous-expression**

Choix conditionnel ? : (2/2)

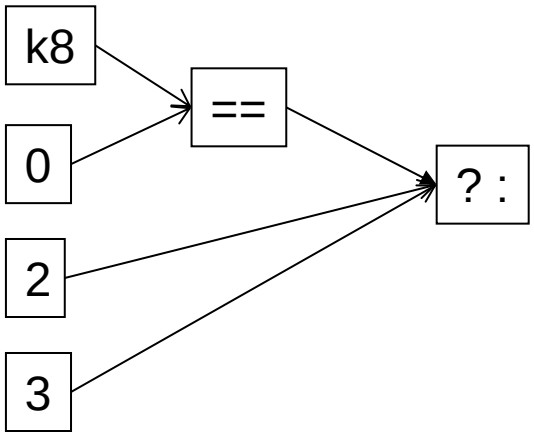
- Le résultat de l'opération est :
  - *valeur2* si *valeur1* est true
  - *valeur3* si *valeur1* est false
- L'opérateur ? : "choisit" donc entre *valeur2* et *valeur3*, selon que *valeur1* est vrai (true) ou faux (false)

? : dans un arbre d'exécution

- Trois flèches entrantes pour le choix conditionnel (? :)
  - Flèche du haut : premier argument
  - Flèche du milieu : deuxième argument
  - Flèche du bas : troisième argument

Exemple 5

k8 == 0 ? 2 : 3



N.B.: On ne peut pas connaître le résultat, car il dépend de la valeur de `k8`

Opérateurs booléens (1/5)

- OU : | |
- ET : & &
- NON : !

- N.B.: OU et ET sont composés de *2 caractères* (comme le ==) qui doivent être collés

Opérateurs booléens (2/5)

- Servent à combiner deux valeurs booléennes (`true` et `false`)
- Permettent (entre autres) d'exprimer des conditions complexes, basées sur *plusieurs comparaisons*

Opérateurs booléens (3/5)

- Exemples :
  - Vérifier si un montant est entre 10 et 100

```
(montant >= 10) && (montant <= 100)
```

  - Vérifier si un caractère est un "V" ou un "F"

```
(car == "V") || (car == "F")
```

Opérateurs booléens (4/5)

- Attention, les valeurs à gauche et à droite doivent être **booléennes**
- On ne peut pas juste écrire :

```
(car == "V" || "F")
```

  - parce que "F" est une valeur caractère, et non booléenne
- ni : `(montant >= 10 && <= 100)`
  - erreur de syntaxe, car pas de valeur à gauche du `<=`

Opérateurs booléens (5/5)

- Peuvent être enchaînés de la façon habituelle (comme les autres opérations)
- Exemple:

```
(car >= 'a' && car <= 'z') ||
(car >= '0' && car <= '9')
```

Exemple 012

Opérateurs à syntaxe spéciale

- Sur valeurs numériques:
  - Math.min(), Math.max(), Math.PI, Math.sqrt()
- Sur valeurs caractères:
  - .length
  - .charAt()
  - .slice()
  - .toUpperCase(), .toLowerCase()
  - .indexOf(), .lastIndexOf()

Terminologie

- Quand un opérateur consiste en un ou des mots qui décrivent l'opération effectuée, on utilise *opérateur* et *opération* de façon interchangeable
  - .length, Math.min(), .indexOf(), .slice(), etc.
- Les opérateurs qui utilisent la syntaxe avec () s'appellent aussi "fonctions"
  - Math.min(), .indexOf(), .slice(), etc.

Convention de notation

- La plupart de ces opérateurs s'utilisent avec des () après leur nom
  - Seules exceptions ici : `Math.PI` et `.length`
- Par convention, pour rappeler qu'elles doivent être suivies de (), on inscrit toujours les parenthèses après leur nom
  - Par exemple : `Math.min()`

N.B.

- Les opérations `alert()` et `prompt()` déjà rencontrées *étaient aussi des fonctions*

Math.min() et Math.max()

- "Calculent" (en fait, trouvent) le minimum ou le maximum de deux valeurs numériques ou plus

```
Math.max(45, 22) → 45
Math.max(22, 45) → 45
Math.min(22, 45) → 22
Math.min(45, 22, -46) → -46
Math.max(0, 10*10, 99) → 100
```

.slice()

- Extrait une sous-chaîne entre deux positions

  `"abcde".slice(1, 3) → "bc"`

  – Positions commencent à 0

  – Position de fin est exclue

Précisions (1/4)

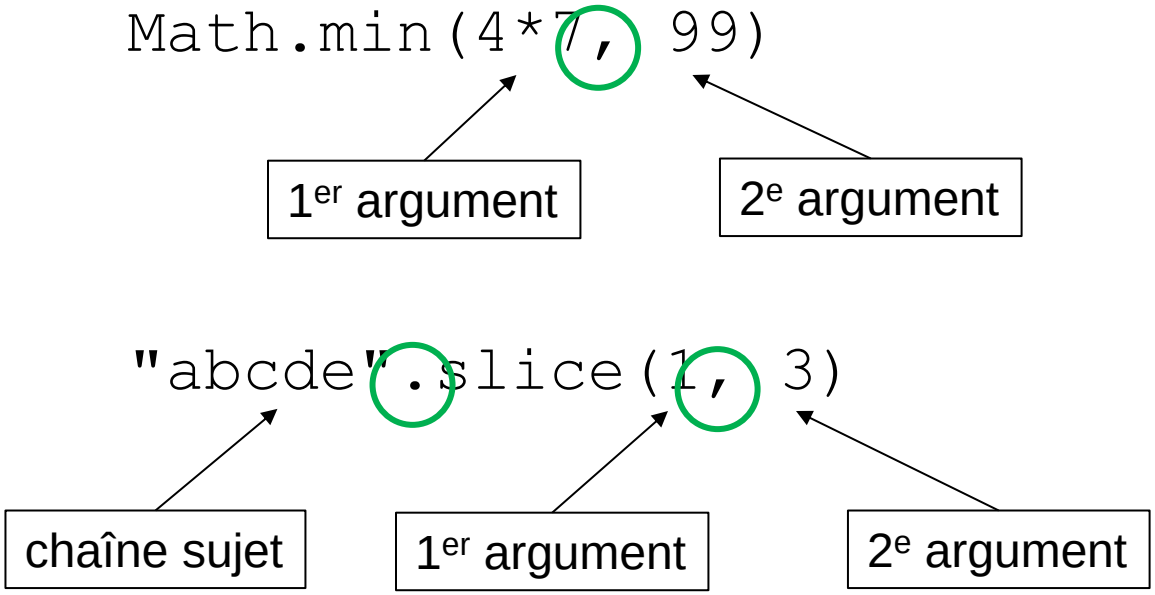
- Avec les opérateurs qui utilisent les ()
  - par exemple `Math.min()` ou `Math.max()`
  - les arguments, séparés par une virgule, sont "numérotés" de gauche à droite : 1e, 2e, etc.
- S'il y a un seul argument, il peut aussi être appelé "premier" argument
  - Par exemple, avec `prompt()`, à venir

Précisions (2/4)

- Les notions d'arguments *de gauche* et *de droite* ne s'appliquent plus
  - Les 2 sont du même côté, entre ( )
- Chaque argument est traité comme une *sous-expression*, indépendante des autres arguments
- Les arguments sont évalués séparément, de gauche à droite

Précisions (3/4)

- Exemples :



Précisions (4/4)

- La virgule qui sépare les arguments d'un appel de fonction n'est pas considérée comme un opérateur, c'est un simple séparateur; par exemple :

```
Math.min(a, b)
```

- Cette virgule n'apparaît pas dans les arbres d'exécution

Opérateurs à syntaxe spéciale et arbres d'exécution

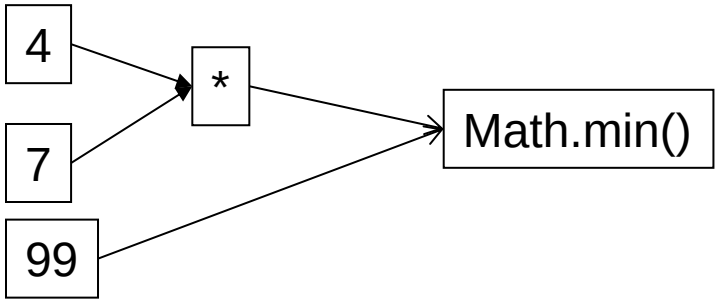
- Tout nom suivi d'une "(" et/ou précédé d'un "." est considéré opérateur, par ex. :

```
.length
.slice()
Math.min()
```

- `Math.PI` est traité comme une constante

Exemple 6

Math.min(4*7, 99)



Résultat : 28

.toUpperCase()

- Transforme une chaîne en majuscules

```
"Il fait beau !".toUpperCase()
  → "IL FAIT BEAU !"
"HA HA HA".toUpperCase()
  → "HA HA HA"
```

Chaîne "sujet" (1/2)

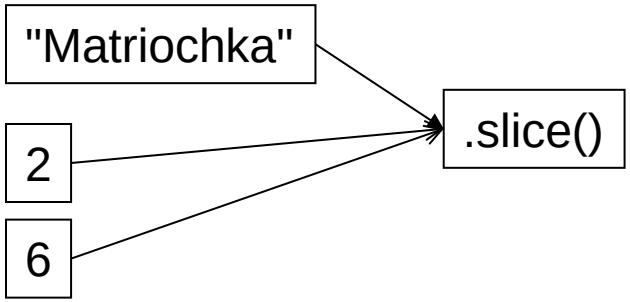
- Avec les opérateurs qui commencent par "." et qui travaillent sur une valeur caractère :
  - La valeur qui précède le point s'appelle le *sujet* ou la *chaîne sujet* de l'opération
  - Les autres valeurs, données entre () après le nom de l'opération, sont les *arguments*
    - Il peut y en avoir 0, 1, ou plus

Chaîne "sujet" (2/2)

- Dans un arbre d'exécution, la chaîne sujet, *comme les autres arguments entre parenthèses* (s'il y en a), est connectée à l'opération par une flèche
- S'il n'y a pas d'arguments entre () après, l'opération, la chaîne sujet est la seule flèche liée a l'opération

Exemple 7

"Matriochka".slice(2, 6)



Résultat : "trio"

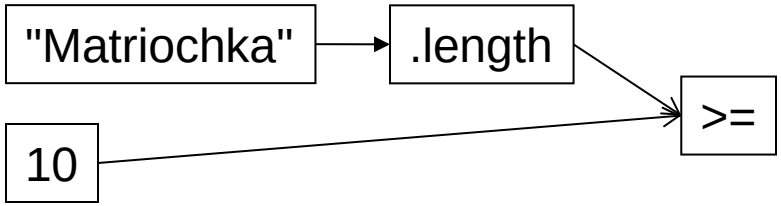
.length

- Donne la longueur de la chaîne sujet

```
"Nonobstant".length → 10
"".length → 0
("a" + "bc" + "d").length → 4
```

Exemple 8

"Matriochka".length >= 10



Résultat : true

.charAt()

- Extrait un caractère à une certaine position dans la chaîne sujet

```
"Kermit".charAt(2) → "r"
"Kermit".charAt(0) → "K"
"Kermit".charAt() → "K"
```

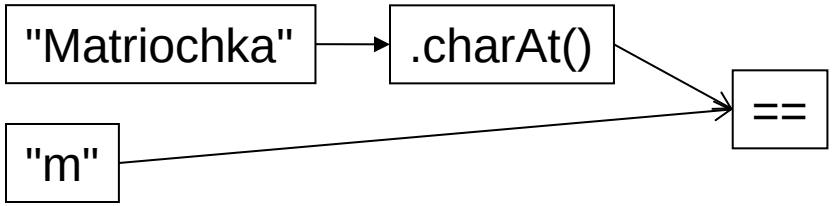
  - Sans argument : comme 0 (1er caractère)

```
"Kermit".charAt(6) → ""
```

  - 6 : position inexistante

Exemple 9

"Matriochka".charAt() == "m"



Résultat : false

.indexOf() et .lastIndexOf() (1/3)

- *chaîne1*.`indexOf`(*chaîne2*)
  - retourne position de la première occurrence de *chaîne2* dans *chaîne1*
  - ou `-1` si la sous-chaîne n'est pas trouvée
- *chaîne1*.`lastIndexOf`(*chaîne2*)
  - retourne position de la *dernière* occurrence de *chaîne2* dans *chaîne1*
  - ou `-1` si la sous-chaîne n'est pas trouvée

.indexOf() et .lastIndexOf() (2/3)

- *chaîne1 et chaîne2* représentent des expressions donnant une valeur caractère
- Comme toujours, origine 0, i.e. le premier caractère est en position 0
- Ex.:

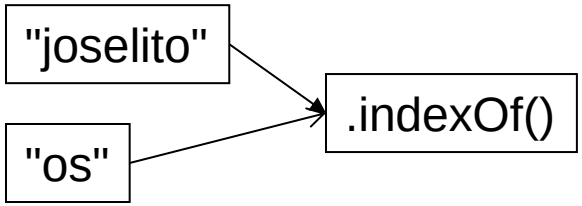
```
"abc".indexOf("bc") → 1
"abcb".lastIndexOf('b') → 3
"abcb".lastIndexOf('bc') → 1
"abc".indexOf("ac") → -1
```

.indexOf() et .lastIndexOf() (3/3)

- *chaîne1* est la chaîne sujet de l'opération
  - C'est la chaîne *interrogée*
- *chaîne2* est donnée en argument
  - C'est la chaîne *recherchée*

Exemple 10

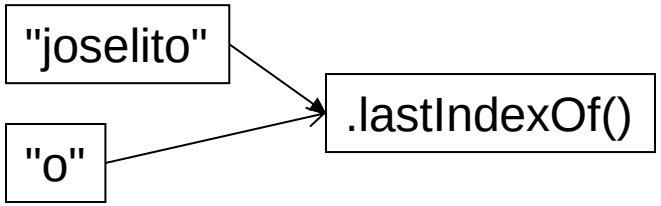
"joselito".indexOf("os")



Résultat : 1

Exemple 11

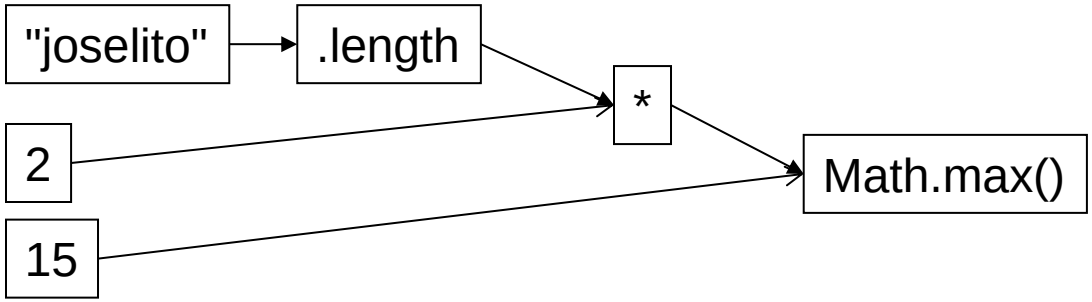
"joselito".lastIndexOf("o")



Résultat : 7

Exemple 12

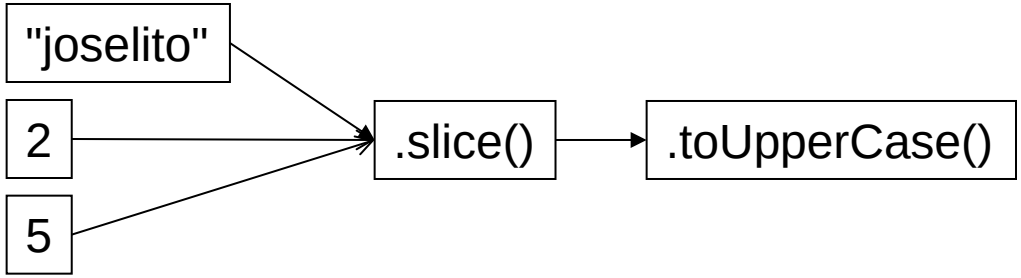
Math.max("joselito".length * 2, 15)



Résultat : 16

Exemple 13

"joselito".slice(2, 5).toUpperCase()



Résultat : "SEL"

Conversions automatiques (3/3)

- 'ab8c'.indexOf(8.0) → 2
- `Math.min()` et `Math.max()` convertissent automatiquement les arguments caractères en valeur numérique
  - Si un (ou plus) des arguments est une chaîne qui ne représente pas un nombre, la fonction retourne : `NaN`

Pour info seulement (PYM) (1/3)

- `prompt()` accepte aussi un 2e argument, aussi de type caractère :
  - C'est une réponse par défaut à mettre dans la zone de saisie
  - L'utilisatrice peut "accepter" la réponse proposée (OK ou Entrée)
  - ou la modifier puis faire OK ou Entrée

Pour info seulement (PYM) (2/3)

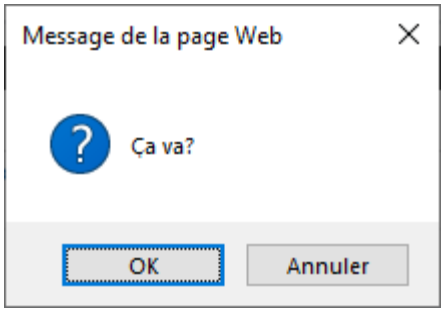
- Autre fonction d'interaction de base

  confirm("Message")

  – Affiche "Message" et offre à l'utilisateur les boutons OK et Annuler

  – Exemple :

```
confirm("Ça va?")
```



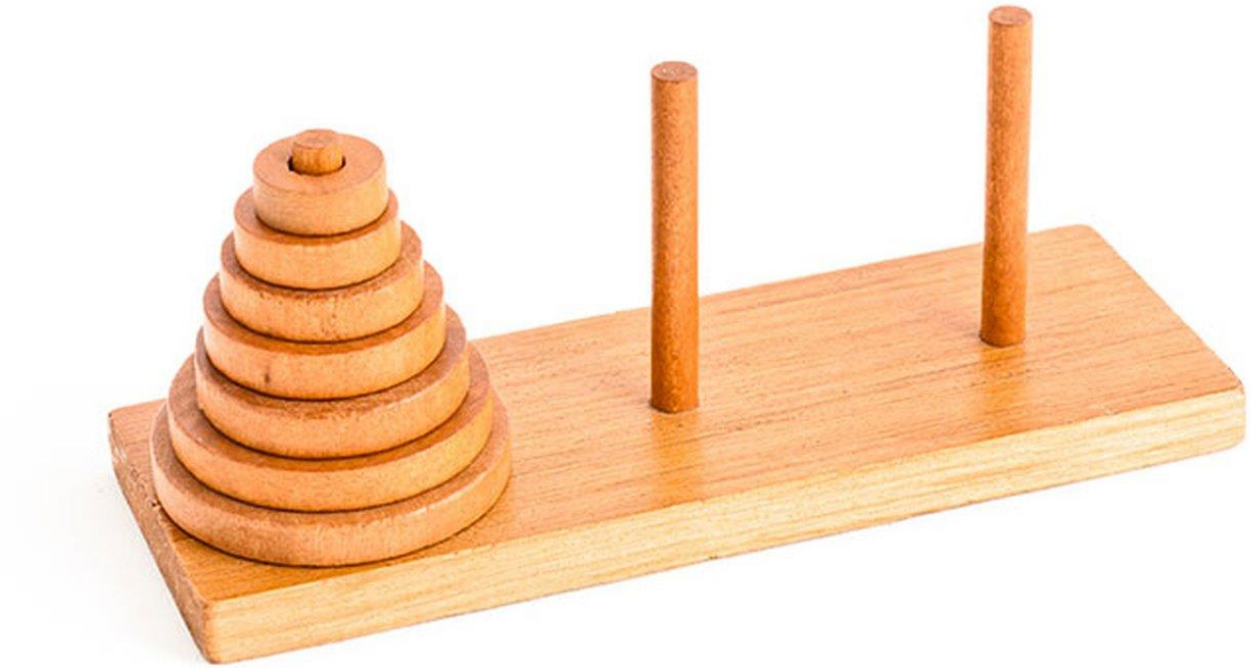
Pour info seulement (PYM) (3/3)

- Le *résultat* de confirm() nous renseigne sur le bouton cliqué par l'utilisatrice
  - valeur `true` si cliqué OK
  - valeur `false` si cliqué Annuler

Pensée *algorithmique*

- Les deux composantes :
  - Décomposer le travail à faire en opérations élémentaires, en fonction du contenu de notre "coffre à outils"
  - Recomposer en un programme exécutable :
    - → Expressions
    - → Énoncés
    - → Structures de contrôle (ordonne les énoncés)

Tours de Hanoï



Méthode de travail (1/2)

- Penser d'abord à la logique générale
  - Faut-il "préparer" les données? Comment?
    - Les découper ?
    - Les normaliser (diminue le nombre de cas) ?
  - Découper le traitement à effectuer
  - C'est l'*approche algorithmique*
- Commencer petit, tests constants
  - Ne pas viser la version finale du premier coup
  - Ajouts très graduels

Méthode de travail (2/2)

- Toujours garder en tête l'inventaire de son "coffre à outils"
  - Le repasser régulièrement face à un traitement spécifique à effectuer

Patrons de conception (PC)

- *Design Patterns*
  - Patron de conception (PC)
  - Façon stéréotypée d'aborder certains traitements à faire
- DNC (diminuer le nombre de cas)
  - "Normaliser" les données est souvent une façon de diminuer le nombre de cas
    - Ex.: transformer une chaîne en majuscules pour éviter de traiter les cas maj. et min. séparément

Éléments HTML `<script>` (1/2)

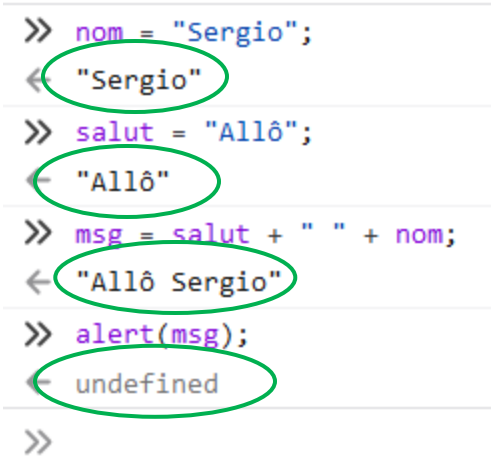
- Peuvent se trouver n'importe où dans une page HTML
- Chacun contient une suite d'*énoncés JS*
  - Exécutés dans leur ordre d'apparition
  - Rappel : *expression JS* + ";" → énoncé
- Les *résultats* des expressions qui, dans la console, s'affichent quand on fait Entrée *ne sont pas affichés*; **ils sont ignorés**

Éléments HTML `<script>` (2/2)

- Il peut y avoir plusieurs `<script>` dans une même page HTML
- Ils sont exécutés dans leur ordre d'apparition dans la page
  - Demeure vrai même quand le code lui-même est dans un fichier distinct (vu plus tard)
- Illustration : exemple 031 (vs 030)

Où vont les résultats des énoncés dans un script ? (1/3)

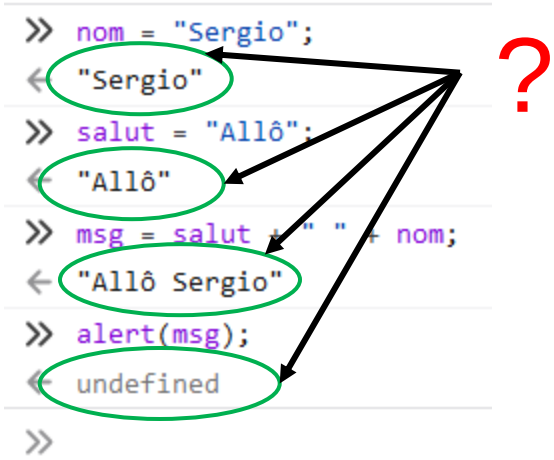
- À la console JS, chaque énoncé retourne un résultat, qui est affiché :



Où vont les résultats des énoncés dans un script ? (2/3)

- Les mêmes énoncés dans un script ne produisent aucun affichage…

```
<script>
nom = "Sergio";
salut = "Allô";
msg = salut + " " + nom;
alert(msg);
</script>
```



Exemple 032

Où vont les résultats des énoncés dans un script ? (3/3)

- Les énoncés dans un script produisent un résultat, mais ces résultats disparaissent simplement et ne sont affichés nulle part
- Ainsi, si on pense faire afficher un message en inscrivant simplement une chaîne de caractères comme énoncé, ça ne marche pas…

Exemple 033

Commentaires JavaScript (1/3)

- Deux formes:
  - 1e forme: Commentaires sur une ligne:
    - Délimité par //
    - Le commentaire va jusqu'à la fin de la ligne
    - Peut être la seule chose sur la ligne
    - Ou être sur la même ligne qu’une instruction
- Exemples:

```
x = x * 1.15;   // Ajouter la taxe
// Modifié par Gaëlle 2027-01-01
```

Commentaires JavaScript (2/3)

- Autre forme:
  - Délimités par /* … */
  - Permet les commentaires sur plusieurs lignes
- Exemple :

```
/* La section qui suit est le calcul
     d'intérêt proprement dit: */
```

Commentaires JavaScript (3/3)

- Les commentaires servent à la documentation pour l'humain ou pour aérer le code (espaces blancs)
- Ils sont complètement ignorés par JS
- Truc de débogage: mettre en commentaires certaines sections problématiques pour les "désactiver" temporairement

Notes sur constantes numériques (1/2)

- L'utilisation de zéros non significatifs *avant le point décimal* est **à proscrire**
  - Peut faire en sorte que le nombre est interprété en octal (si aucun chiffre > 7)
  - Avec le point décimal : cause une erreur !!
- Exemples à essayer en console JS:

  012345

  00.34
- Empêché par `"use strict"` (à venir)

Notes sur constantes numériques (2/2)

- Signes - et + peuvent faire partie d'une constante numérique :

  +'-45' → -45

  +'- 45' → NaN

  +'+45' → 45

  +'+ 45' → NaN

- Notation scientifique ($\times 10^n$):
  - 3.456e3 → 3456
  - -1.034e+6 → -1034000
  - 1e-6 → 0.000001

Caractères spéciaux dans les constantes caractères (1/2)

- Caractère `\n` → saut de ligne
  - Utile avec prompt(), alert() et confirm()
    - Le sera aussi avec les autres modes d'interaction
  - Permet d'insérer un saut de ligne dans le message affiché. Ex.:
    - alert("Bonjour " + nom + ".\nÇa va?");
- `\t` : tabulation peut aussi être utile
- `\"`, `\\` → pour échapper " et \

Caractères spéciaux dans les constantes caractères (2/2)

- On peut écrire une constante caractère sur plusieurs lignes en mettant un \ *avant* le saut de ligne:

```
longueChaine = "Ceci est une lon\
gue chaîne";
longueChaine == "Ceci est une longue chaîne";
    true
```

- Comme si le saut de ligne n'y était pas

Délimiteur d'énoncés

- Le ";" indique la fin d'une instruction (ou énoncé) JavaScript
- Permet de mettre plus d'un énoncé sur la même ligne
  - Attention, pas toujours une bonne idée !
- Rarement obligatoire, mais ne nuit pas
- C'est une bonne pratique: indique explicitement où l'énoncé se termine

Espaces et sauts de ligne (1/2)

- En général, l'espacement et les sauts de ligne n'ont pas de signification
  - Sauf dans les constantes de type caractère
- Sont interdits à certains rares endroits, p.ex.:
  - Entre les "=" des opérateurs "==" et "==="
  - et d'autres opérateurs similaires (||, &&, …)

Où on s'en va ?

- Trois nouvelles sections à lire : 9 à 11
  - Simplement garnir un peu notre coffre à outils
  - Et commencer à s'initier à l'univers conceptuel des objets
- Exercices après C2
  - Le no 4b est l'exercice rapide pour demain matin (5%) !