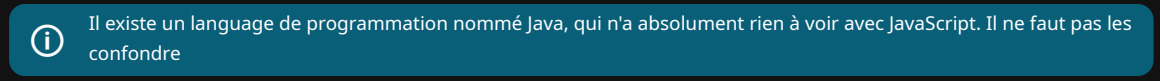

JS

JavaScript

Préambule

- Le JavaScript est créé en 1995
- Standardisé sous le nom d'*ECMAScript*
- Depuis 2015, une nouvelle version sort chaque année
- Parmi les languages les plus utilisés au monde
- Le web utilise ce language pour scripter ses pages, tous les navigateurs savent interpréter du JavaScript



JavaScript

Les ressources

W3Schools : https://www.w3schools.com/js/
Références et tutoriels facile d'accès. Excellent pour débuter

MDN : https://developer.mozilla.org/en-US/docs/Web/JavaScript
Documentation officielle de Mozilla. Référence la plus complète et la plus précise disponible

De nombreux tutoriels et documentations sont également disponibles sur presque tous les supports (livres, sites internet, vidéos, etc.)

N'hésitez pas à chercher dès que vous avez une questions. *Savoir utiliser une documentation doit faire partie de vos compétences*

JavaScript

Où l'écrire

Le javascript s'écrit toujours dans une balise `<script>`. Il peut s'écrire :

A même le HTML

```html
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript</title>
    </head>
    <body>
        <script>
            console.log('Hello World!');
        </script>
    </body>
</html>
```

Dans un fichier externe

```html
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript</title>
    </head>
    <body>
        <script src="script.js"></script>
    </body>
</html>
```

Ces deux codes sont équivalents si `script.js` contient `console.log('Hello World!');`

JavaScript

Ordre d'execution

Le JavaScript s'execute dans l'ordre où la page est lue par le navigateur. Cela implique qu'un script ne peut accéder qu'aux éléments déjà analysés.

Par défaut, l'execution du JavaScript est bloquante pour le chargement de la page. On placera généralement la balise `<script>` à la fin de la balise `<body>` pour ne pas ralentir le chargement.

```
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript</title>
        <script> ... </script>      -> S'exécutera avant le chargement du body, il ne pourra pas y accéder
    </head>
    <body>
        <script> ... </script>      -> S'exécutera après le chargement du body et avant celui du h1
        <h1> Titre </h1>
        <script> ... </script>      -> S'exécutera une fois tous les éléments chargés
    </body>
</html>
```

Cet ordre d'execution est également valable pour les balises de script qui incluent un fichier externe. Cela ne fait pas de différence.

Il existe en JavaScript des événements qui permettent d'executer du code à un moment donné (lorsque la page s'est chargée, lors du clic sur un bouton, quand une ressource est disponible, etc). Ces derniers permettent aussi de controller l'ordre d'execution. Nous en verrons certains plus loins dans le cours.

JavaScript

Les variables

JavaScript utilise un *typage implicite*. Comme en Python, on ne déclare pas le type des variables. Les variables sont déclarées avec `let` , ou avec `const` si leur valeur ne change pas.

```
let maVariable = 3;    // Déclare une variable, sa valeur pourra changer
const maVariable = 3;  // Déclare une constante, changer sa valeur provoquera une erreur
```

Selon la norme ECMAScript, toutes les instructions en JavaScript doivent se terminer par un point-virgule, mais l'interpréteur acceptera si vous ne les mettez pas. Prenez l'habitude de respecter le standard et de toujours les utiliser.

Il est aussi recommandé de terminer toutes les lignes par un point-virgule car cela permet d'éviter des erreurs subtiles de syntaxe.

Par exemple, le code suivant :

```
const a = 1 + 2
("a" + "B").toLowerCase()
```

Retournera l'erreur : *Uncaught TypeError: 2 is not a function*

Car JavaScript interprète un retour à la ligne comme une fin d'instruction **uniquement** quand cela est la seule solution syntaxique.

Or la ligne *`const a = 1 + 2("a" + "B").toLowerCase()`* est syntaxiquement valide, et donc interprétée comme telle. Mais elle retourne une erreur lors de l'execution car "2" suivit de parenthèses ne veut rien dire. Avec des points-virgules cette erreur ne serait pas arrivée.

JavaScript

Les variables

JavaScript utilise un *typage implicite*. Comme en Python, on ne déclare pas le type des variables. Les variables sont déclarées avec `let` , ou avec `const` si leur valeur ne change pas.

```
let maVariable = 3;    // Déclare une variable, sa valeur pourra changer
const maVariable = 3;  // Déclare une constante, changer sa valeur provoquera une erreur
```

> Selon la norme ECMAScript, toutes les instructions en JavaScript doivent se terminer par un point-virgule, mais l'interpréteur acceptera si vous ne les mettez pas. Prenez l'habitude de respecter le standard et de toujours les utiliser.

Il est possible de déclarer des variable avec `var` ou sans utiliser de préfixe. Cela déclare une variable globale au contexte courant. On ne l'utilise plus en pratique, cela créé des confusions.

```
maVariable = 3;       // Ces deux lignes font la même chose. maVariable sera globale à la fonction courante
var maVariable = 3;   // Ces syntaxes sont désuètes, à ne pas utiliser !
```

Il est aussi recommandé de terminer toutes les lignes par un point-virgule car cela permet d'éviter des erreurs subtiles de syntaxe.
Par exemple, le code suivant :

```
const a = 1 + 2
("a" + "B").toLowerCase()
```

Retournera l'erreur : *Uncaught TypeError: 2 is not a function*
Car JavaScript interprète un retour à la ligne comme une fin d'instruction **uniquement** quand cela est la seule solution syntaxique.
Or la ligne *`const a = 1 + 2("a" + "B").toLowerCase()`* est syntaxiquement valide, et donc interprétée comme telle. Mais elle retourne une erreur lors de l'execution car "2" suivit de parenthèses ne veut rien dire. Avec des points-virgules cette erreur ne serait pas arrivée.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

L'instruction `typeof` retourne le type de la variable sous forme de chaîne de caractères

```
const maVariable = false;
const typeDeMaVariable = typeof maVariable; // typeDeMaVariable vaut "boolean"
```

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

Les types

Il existe 7 types de base en JavaScript :

```
const maVariable = "JavaScript";    // Chaîne de caractères
const maVariable = 3.14;            // Nombre réel
const maVariable = true;            // Booléen
const maVariable = null;            // null, indique l'absence d'un objet
const maVariable;                   // undefined, indique qu'il n'y a pas de valeur, "non déclaré"
const maVariable = BigInt(3);       // BigInt, entier sans limites
const maVariable = Symbol("foo");   // Symbol, pas utile dans ce cours
```

L'instruction `typeof` retourne le type de la variable sous forme de chaîne de caractères

```
const maVariable = false;
const typeDeMaVariable = typeof maVariable; // typeDeMaVariable vaut "boolean"
```

Il existe également d'autres types non basiques, qui sont des structures stockant des types basiques (tableaux, objets, etc).

JavaScript est un language à **typage dynamique implicite**
Implicite signifie que le language infère le type selon le contexte, on n'indique jamais explicitement le type d'une variable
Dynamique signifie que la vérification des types est faite lors de l'execution. Par opposition a des languages tels que le C/C++ où la vérification des types est faites lors de la compilation.

Même si on ne déclare pas explicitement le type d'une variable, JavaScript assigne en interne un type à chaque variable.

JavaScript

La console

Pour afficher un texte dans la console :

```
console.log("Bonjour"); // Affichera "Bonjour" dans la console
```

> La fonction print() existe en JavaScript, mais il s'agit de la commande pour imprimer la page web !

Il existe les fonctions pour afficher des messages d'avertissement et d'erreur :

```
console.warn("Attention");
console.error("Erreur");
```

Ces fonctions peuvent afficher n'importe quelle variable de n'importe quel type, profitez-en !

A part les types de bases, tous les autres sont des "objects". Cela veut dire que tous les types de JavaScript suivent une logique similaire et peuvent facilement être affichés dans la console. La console des navigateur, quand un objet y est affiché, ajoute même une interface pour naviguer dans les propriétés de l'objet. Cela est extrêmement pratique pour debugger.

JavaScript

Les opérations booléennes

Les comparaisons classiques sont utilisées :

```
console.log(3 < 2);    // false
console.log(3 >= 2);    // true
console.log(3 == 2);   // false
console.log(3 != 2);   // false
```

JavaScript

Les opérations booléennes

Les comparaisons classiques sont utilisées :

```
console.log(3 < 2);    // false
console.log(3 >= 2);    // true
console.log(3 == 2);   // false
console.log(3 != 2);   // false
```

Les opérateurs `===` et `!==` permettent de tester la valeur et le type :

```
console.log(2 == "2");    // true      car les valeurs sont identiques, peu importe le type
console.log(2 === "2");   // false     car les types sont différents
console.log(2 != "2");    // false
console.log(2 !== "2");   // true
```

JavaScript

Les opérations booléennes

Les comparaisons classiques sont utilisées :

```
console.log(3 < 2);    // false
console.log(3 >= 2);    // true
console.log(3 == 2);   // false
console.log(3 != 2);   // false
```

Les opérateurs `===` et `!==` permettent de tester la valeur et le type :

```
console.log(2 == "2");    // true      car les valeurs sont identiques, peu importe le type
console.log(2 === "2");   // false     car les types sont différents
console.log(2 != "2");    // false
console.log(2 !== "2");   // true
```

Les opérateurs `&&`, `||` et `!` fonctionnent respectivement comme `and`, `or` et `not` en Python

```
console.log( !(2 != "2" || 3 < 2) && 0 < 4); // true
```

JavaScript

Les opérations mathématiques

En JavaScript on retrouve les opérations des base : addition `+`, soustraction `-`, multiplication `*`, division `/`, modulo `%` (reste d'une division entière).

Il existe également les opérateurs d'assignement : `+=`, `-=`, `*=`, `/=`
`x += y` est équivalent à `x = x + y`, idem pour les autres opérateurs

Il existe aussi les opérateurs d'incrémentation `++` et de décrémentation `--`
`x++` est équivalent à `x = x + 1` et `x--` est équivalent à `x = x - 1`

L'opérateur `+` sert de concaténation pour les chaînes de caractères.
`"Hello " + "World"` donne `"Hello World"`

Pour tous les détails, voir https://www.w3schools.com/js/js_operators.asp

JavaScript

Les fonctions mathématiques

Toutes les fonctions de math avancées se font grâce à l'objet `Math`

- `Math.abs(x)` : valeur absolue
- `Math.log(x)`, `Math.exp(x)` : logarithme et exponentielle
- `Math.min(x, y, z, ...)`, `Math.max(x, y, z, ...)` : minimum et maximum de plusieurs valeurs
- `Math.random()` : nombre aléatoire dans l'intervalle `[0, 1[`
- `Math.sqrt(x)`, `Math.pow(x, y)` : racine carrée et puissance
- `Math.round(x)`, `Math.floor(x)`, `Math.ceil(x)` : Arrondi resp. au plus proche, inférieur, supérieur
- `Math.PI` : π

Référence complète :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Math

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);       // 8        Addition standard
console.log("3" + "5");   // "35"     Concaténation de texte
console.log(3 + "5");     // "35"     Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);       // -2       Soustraction standard
console.log("3" - "5");   // -2     La soustraction converti automatique tout texte en nombre
console.log("3" - "A");   // NaN      Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");   // 15       Idem pour tous les autres opérateurs
console.log("3" * true);  // 3        true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);  // 3        null est converti en la valeur 0
console.log("3" - []);    // 3        Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);       // 8         Addition standard
console.log("3" + "5");   // "35"      Concaténation de texte
console.log(3 + "5");     // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);       // -2        Soustraction standard
console.log("3" - "5");   // -2      La soustraction converti automatique tout texte en nombre
console.log("3" - "A");   // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");   // 15        Idem pour tous les autres opérateurs
console.log("3" * true);  // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);  // 3         null est converti en la valeur 0
console.log("3" - []);    // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);         // 8         Addition standard
console.log("3" + "5");     // "35"      Concaténation de texte
console.log(3 + "5");       // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);         // -2        Soustraction standard
console.log("3" - "5");     // -2      La soustraction converti automatique tout texte en nombre
console.log("3" - "A");     // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");     // 15        Idem pour tous les autres opérateurs
console.log("3" * true);    // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);    // 3         null est converti en la valeur 0
console.log("3" - []);      // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN     Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);        // 8         Addition standard
console.log("3" + "5");    // "35"      Concaténation de texte
console.log(3 + "5");      // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);        // -2        Soustraction standard
console.log("3" - "5");    // -2      La soustraction converti automatique tout texte en nombre
console.log("3" - "A");    // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");    // 15        Idem pour tous les autres opérateurs
console.log("3" * true);   // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);   // 3         null est converti en la valeur 0
console.log("3" - []);     // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN   Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);        // 8         Addition standard
console.log("3" + "5");    // "35"      Concaténation de texte
console.log(3 + "5");      // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);        // -2        Soustraction standard
console.log("3" - "5");    // -2     La soustraction converti automatique tout texte en nombre
console.log("3" - "A");    // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");    // 15        Idem pour tous les autres opérateurs
console.log("3" * true);   // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);   // 3         null est converti en la valeur 0
console.log("3" - []);     // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);         // 8         Addition standard
console.log("3" + "5");     // "35"      Concaténation de texte
console.log(3 + "5");       // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);         // -2        Soustraction standard
console.log("3" - "5");     // -2      La soustraction converti automatique tout texte en nombre
console.log("3" - "A");     // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");     // 15        Idem pour tous les autres opérateurs
console.log("3" * true);    // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);    // 3         null est converti en la valeur 0
console.log("3" - []);      // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN     Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);        // 8         Addition standard
console.log("3" + "5");    // "35"      Concaténation de texte
console.log(3 + "5");      // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);        // -2        Soustraction standard
console.log("3" - "5");    // -2     La soustraction converti automatique tout texte en nombre
console.log("3" - "A");    // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");    // 15        Idem pour tous les autres opérateurs
console.log("3" * true);   // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);   // 3         null est converti en la valeur 0
console.log("3" - []);     // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);        // 8         Addition standard
console.log("3" + "5");    // "35"      Concaténation de texte
console.log(3 + "5");      // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);        // -2        Soustraction standard
console.log("3" - "5");    // -2     La soustraction converti automatique tout texte en nombre
console.log("3" - "A");    // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");    // 15        Idem pour tous les autres opérateurs
console.log("3" * true);   // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);   // 3         null est converti en la valeur 0
console.log("3" - []);     // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);        // 8         Addition standard
console.log("3" + "5");    // "35"      Concaténation de texte
console.log(3 + "5");      // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);        // -2        Soustraction standard
console.log("3" - "5");    // -2     La soustraction converti automatique tout texte en nombre
console.log("3" - "A");    // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");    // 15        Idem pour tous les autres opérateurs
console.log("3" * true);   // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);   // 3         null est converti en la valeur 0
console.log("3" - []);     // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);        // 8         Addition standard
console.log("3" + "5");    // "35"      Concaténation de texte
console.log(3 + "5");      // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);        // -2        Soustraction standard
console.log("3" - "5");    // -2     La soustraction converti automatique tout texte en nombre
console.log("3" - "A");    // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");    // 15        Idem pour tous les autres opérateurs
console.log("3" * true);   // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);   // 3         null est converti en la valeur 0
console.log("3" - []);     // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);        // 8         Addition standard
console.log("3" + "5");    // "35"      Concaténation de texte
console.log(3 + "5");      // "35"      Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);        // -2        Soustraction standard
console.log("3" - "5");    // -2     La soustraction converti automatique tout texte en nombre
console.log("3" - "A");    // NaN       Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");    // 15        Idem pour tous les autres opérateurs
console.log("3" * true);   // 3         true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);   // 3         null est converti en la valeur 0
console.log("3" - []);     // 3         Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);       // 8        Addition standard
console.log("3" + "5");   // "35"     Concaténation de texte
console.log(3 + "5");     // "35"     Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);       // -2       Soustraction standard
console.log("3" - "5");   // -2     La soustraction converti automatique tout texte en nombre
console.log("3" - "A");   // NaN      Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");   // 15       Idem pour tous les autres opérateurs
console.log("3" * true);  // 3        true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);  // 3        null est converti en la valeur 0
console.log("3" - []);    // 3        Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

JavaScript

Coercition

Les opérateurs mathématiques peuvent être utilisés entre tout types, mais attention aux comportements surprenants ! L'addition est particulière.

```
console.log(3 + 5);       // 8        Addition standard
console.log("3" + "5");   // "35"     Concaténation de texte
console.log(3 + "5");     // "35"     Dès qu'un opérateur est un texte, c'est une concaténation
console.log(3 - 5);       // -2       Soustraction standard
console.log("3" - "5");   // -2     La soustraction converti automatique tout texte en nombre
console.log("3" - "A");   // NaN      Si un texte n'est pas convertible, la valeur NaN (Not A Number est retournée)
console.log("3" * "5");   // 15       Idem pour tous les autres opérateurs
console.log("3" * true);  // 3        true est converti en la valeur 1, false est converti en la valeur 0
console.log("3" - null);  // 3        null est converti en la valeur 0
console.log("3" - []);    // 3        Les tableau vides valent 0
console.log("3" - undefined); // NaN  Une opération avec "undefined" provoquera toujours un Nan
```

Il existe beaucoup d'autres cas. On apelle cette "conversion automatique" entre types la *coercition*.

Retenez : Éviter au maximum les opérations entres types différents !

JavaScript

Cas particuliers

Pas convaincu de ne pas mélanger les types ?

```
console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // Qu'affiche ceci ?
```

On entend beaucoup de critiques de JavaScript à cause de ces comportements bizarres. Pour nuancer un peu, il s'agit la majorité du temps de codes qui ne s'écrivent pas en pratique, ou de cas poussés à l'extrême.

N'oubliez pas : gardez toujours votre code propre et clair. En faisant ça vous évitez automatiquement ce genre de problèmes.

JavaScript

Cas particuliers

Pas convaincu de ne pas mélanger les types ?

```
console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // Qu'affiche ceci ?
```

```
console.log(+ 'lancer');                                  // NaN      car le + converti le 'lancer' en nombre
```

On entend beaucoup de critiques de JavaScript à cause de ces comportements bizarres. Pour nuancer un peu, il s'agit la majorité du temps de codes qui ne s'écrivent pas en pratique, ou de cas poussés à l'extrême.

N'oubliez pas : gardez toujours votre code propre et clair. En faisant ça vous évitez automatiquement ce genre de problèmes.

JavaScript

Cas particuliers

Pas convaincu de ne pas mélanger les types ?

```
console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // Qu'affiche ceci ?
```

```
console.log(+ 'lancer');                                   // NaN      car le + converti le 'lancer' en nombre
```

```
console.log('a' + (+'lancer'));                            // aNaN     car un + avec du texte converti en texte
```

On entend beaucoup de critiques de JavaScript à cause de ces comportements bizarres. Pour nuancer un peu, il s'agit la majorité du temps de codes qui ne s'écrivent pas en pratique, ou de cas poussés à l'extrême.

N'oubliez pas : gardez toujours votre code propre et clair. En faisant ça vous évitez automatiquement ce genre de problèmes.

JavaScript

Cas particuliers

Pas convaincu de ne pas mélanger les types ?

```
console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // Qu'affiche ceci ?
```

```
console.log(+ 'lancer');                                    // NaN      car le + converti le 'lancer' en nombre
```

```
console.log('a' + (+'lancer'));                             // aNaN     car un + avec du texte converti en texte
```

```
console.log('a' + + 'lancer');                              // aNaN     car JavaScript suit la priorité des opérations
```

On entend beaucoup de critiques de JavaScript à cause de ces comportements bizarres. Pour nuancer un peu, il s'agit la majorité du temps de codes qui ne s'écrivent pas en pratique, ou de cas poussés à l'extrême.

N'oubliez pas : gardez toujours votre code propre et clair. En faisant ça vous évitez automatiquement ce genre de problèmes.

JavaScript

Cas particuliers

Pas convaincu de ne pas mélanger les types ?

```
console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // Qu'affiche ceci ?
```

```
console.log(+ 'lancer');                                    // NaN      car le + converti le 'lancer' en nombre
```

```
console.log('a' + (+'lancer'));                             // aNaN     car un + avec du texte converti en texte
```

```
console.log('a' + + 'lancer');                              // aNaN     car JavaScript suit la priorité des opérations
```

```
console.log('a' + + 'lancer' + 'a');                        // aNaNa
```

On entend beaucoup de critiques de JavaScript à cause de ces comportements bizarres. Pour nuancer un peu, il s'agit la majorité du temps de codes qui ne s'écrivent pas en pratique, ou de cas poussés à l'extrême.

N'oubliez pas : gardez toujours votre code propre et clair. En faisant ça vous évitez automatiquement ce genre de problèmes.

JavaScript

Cas particuliers

Pas convaincu de ne pas mélanger les types ?

```
console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // Qu'affiche ceci ?

console.log(+ 'lancer');                                   // NaN      car le + converti le 'lancer' en nombre

console.log('a' + (+'lancer'));                            // aNaN     car un + avec du texte converti en texte

console.log('a' + + 'lancer');                             // aNaN     car JavaScript suit la priorité des opérations

console.log('a' + + 'lancer' + 'a');                       // aNaNa

console.log(('b' + 'a' + + 'lancer' + 'a'));               // baNaNa
```

On entend beaucoup de critiques de JavaScript à cause de ces comportements bizarres. Pour nuancer un peu, il s'agit la majorité du temps de codes qui ne s'écrivent pas en pratique, ou de cas poussés à l'extrême.

N'oubliez pas : gardez toujours votre code propre et clair. En faisant ça vous évitez automatiquement ce genre de problèmes.

JavaScript

Cas particuliers

Pas convaincu de ne pas mélanger les types ?

```
console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // Qu'affiche ceci ?

console.log(+ 'lancer');                                   // NaN      car le + converti le 'lancer' en nombre

console.log('a' + (+'lancer'));                            // aNaN     car un + avec du texte converti en texte

console.log('a' + + 'lancer');                             // aNaN     car JavaScript suit la priorité des opérations

console.log('a' + + 'lancer' + 'a');                       // aNaNa

console.log(('b' + 'a' + + 'lancer' + 'a'));               // baNaNa

console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // banana   toLowerCase met tout en minuscules
```

On entend beaucoup de critiques de JavaScript à cause de ces comportements bizarres. Pour nuancer un peu, il s'agit la majorité du temps de codes qui ne s'écrivent pas en pratique, ou de cas poussés à l'extrême.

N'oubliez pas : gardez toujours votre code propre et clair. En faisant ça vous évitez automatiquement ce genre de problèmes.

JavaScript

Cas particuliers

Pas convaincu de ne pas mélanger les types ?

```
console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // Qu'affiche ceci ?

console.log(+ 'lancer');                                   // NaN      car le + converti le 'lancer' en nombre

console.log('a' + (+'lancer'));                            // aNaN     car un + avec du texte converti en texte

console.log('a' + + 'lancer');                             // aNaN     car JavaScript suit la priorité des opérations

console.log('a' + + 'lancer' + 'a');                       // aNaNa

console.log(('b' + 'a' + + 'lancer' + 'a'));               // baNaNa

console.log(('b' + 'a' + + 'lancer' + 'a').toLowerCase()); // banana   toLowerCase met tout en minuscules
```

Liste d'opérations WTF en JavaScript: https://github.com/denysdovhan/wtfjs

On entend beaucoup de critiques de JavaScript à cause de ces comportements bizarres. Pour nuancer un peu, il s'agit la majorité du temps de codes qui ne s'écrivent pas en pratique, ou de cas poussés à l'extrême.

N'oubliez pas : gardez toujours votre code propre et clair. En faisant ça vous évitez automatiquement ce genre de problèmes.

JavaScript

Les conditions

Les conditions s'écrivent de la manière suivante :

```
if(a < b){
    console.log("Do A");
}else if(a < c){
    console.log("Do B");
}else{
    console.log("Do C");
}
```

Comme en Python, les blocs `else if` et `else` sont facultatifs

Il est aussi possible de faire des ternaires avec la syntaxe `<condition> ? <si_vrai> : <si_faux>`

```
const a = 5;
const value = a > 3 ? "Plus grand" : "Plus petit ou égal";
console.log(value); // "Plus grand"
```

Détail concernant les conditions: JavaScript (comme Python, C/C++ et bien d'autres) évalue les conditions en court-circuit. Cela veut dire que JavaScript évalue les conditions élément par élément, et continue l'execution dès qu'une décision peut être prise, même si toute l'expression n'a pas été évaluée.
Exemple :

```
const a = [];
// Cette ligne est valide, car comme "a.length <= 0" est vrai, la condition s'execute sans évaluer la suite
if(a.length <= 0 || a[0][0] == 2){ console.log("Valide"); }

// Cette ligne provoque une erreur, car "a[0]" est undefined et donc "a[0][0]" est invalide
if(a[0][0] == 2 || a.length <= 0){ console.log("Erreur"); }
```

JavaScript

Les boucles – la boucle while

Pour répéter des instructions plusieurs fois, il existe les boucles :

```
let text = "";
while(text != "Bonjour"){
    text = prompt("Dites 'Bonjour' !");  // prompt() Demande à l'utilisateur d'entrer un texte
}
console.log("Vous avez dit bonjour !");
```

La boucle `while (condition) {instructions}` répète les `instructions` tant que `condition` est vraie.

La boucle `while` est utile quand *on ne connait pas le nombre de répétitions à faire*.

Attention à *toujours* vous assurer que la boucle se terminera, sinon c'est une boucle infinie

L'évaluation de la condition est faite au début de chaque itération

Les boucle infinie sont particulièrement dangereuses care elles bloquent complètement la page, et il n'y a pas d'autres choix que de recharger la page.

S'il n'est pas évident que la boucle s'arrêtera quoi qu'il arrive, pensez à mettre une condition d'arrêt, ou utiliser une boucle *for* à la place.

JavaScript

Les boucles – la boucle for

Pour répéter des instructions plusieurs fois, il existe les boucles :

```
for(let i=0; i<10; i++){
    console.log(i); // 0 1 2 3 4 5 6 7 8 9
}
```

La boucle `for (initialisateur; condition; iteration) {instructions}` :

- Execute `initialisateur` avant d'entrer dans la boucle
- Execute `iteration` après chaque tour de boucle
- S'execute tant que `condition` est vraie

La boucle `for` est utile quand *on connait le nombre de répétitions à faire*. Elle permet d'éviter des erreurs qui créent des bouclent infinies, car il est plus simple de voir que la boucle s'arrêter quoi qu'il arrive.

Toute boucle *for* peut être écrite sous forme de boucle *while* et inversement. Choisir l'une ou l'autre dépend du contexte. On préférera le type de boucle qui permet d'exprimer le plus simplement la logique du code.

JavaScript

Les boucles – break & continue

Dans une boucle, l'instruction `break` sort immédiatement de la boucle, sans condition.

```
const a = [/*Quelque chose*/];
for(let i=0; i<a.length; i++){
    if (a[i] == 0) {            // Si a[i] vaut zéro, sort de la boucle
        break;
    }
    console.log(`${1/a[i]}`);   // Ceci affichera l'inverse de tous les éléments du tableau, jusqu'au premier zéro
}
```

Dans une boucle, l'instruction `continue` passe immédiatement à l'itération suivante.

```
const a = [/*Quelque chose*/];
for(let i=0; i<a.length; i++){
    if (a[i] == 0) {            // Si a[i] vaut zéro, ignore l'itération actuelle et continue la boucle
        continue;
    }
    console.log(`${1/a[i]}`);   // Ceci affichera l'inverse de tous les éléments du tableau, sauf ceux valant zéro
}
```

Les instructions *break* et *continue* affectent toujours la boucle courante (la boucle la plus intérieure dans le cas de boucle imbriquées).

Toutes les boucles utilisant *break* et *continue* peuvent se réécrire sans ces mots clés.
Comme pour les choix entre *while* et *for*, il faut préférer l'option qui produit le code le plus facilement compréhensible.

JavaScript

Les tableaux

Les tableaux sont l'équivalent des listes en Python

```
const a = [];                           //  Tableau vide
const b = [1, 2, 3];                    //  Tableau de nombres
const c = [1, "a", null, 12.4, [1,2] ]; //  Les tableau peuvent contenir n'importe quel mélange de types
console.log(c[2]);                      //  Accès aux éléments : affiche null. La numérotation commence toujours à 0
c[58] = true;                           // Il est possible d'assigner n'importe quel élément d'un tableau.
                                        // Tous les élément non assignés valent 'undefined'


b.push(5);                              //  "push" pour ajouter un élément, b contient alors [ 1, 2, 3, 5 ]
b.pop();                                //  "pop" supprime et renvoie le dernier élément, b contient alors  [1, 2, 3]

console.log(b.length);                  // Affiche 3. La propriété "length" retourne la taille du tableau

const d = [[1,2,3], [4,5,6], [7,8,9]];  // Il est possible d'avoir des tableaux de tableaux
console.log(d[1][2]);                   // Affiche 6
```

Techniquement, en JavaScript les tableaux sont des objets. `typeof []` retourne `"object"`

En incluant des tableaux dans des tableaux, il est possible de créer des tableaux avec n'importe quelle dimension. Par exemple

```
const m = [
    [ [1,2],  [3, 4]  ],
    [ [5,6],  [7, 8]  ],
    [ [9,10], [11,12] ]
];
```

est un tableau à 3 dimensions auquel on accédera, par exemple, par *m[2][1][0]*.

JavaScript

Les fonctions des tableaux

- Taille d'un tableau : `array.length`
- Ajouter un élément à la fin: `array.push(2)`
- Retirer le dernier élément : `array.pop()` Renvoi l'élément retiré
- Trier le tableau : `array.sort()` Trie le tableau et le renvoie
- Inverser l'ordre des éléments : `array.reverse()` Inverse l'ordre des éléments et renvoie le tableau
- Appartenance : `array.includes('value')` Retourne `true` si le tableau contient 'value', `false` sinon

Et bien d'autres : https://www.w3schools.com/js/js_array_methods.asp

JavaScript

Les objets

Un objet en JavaScript est équivalent à un dictionnaire en Python

```
const object = {
    'key1' : 3,
    'key2' : "Value",
    'key3' : {
        'subkey1' : true,
        'subkey2' : 5
    },
    'key4' : [4,5,6]
};
```

Un objet est un ensemble de valeurs où chacune possède une clé. On accède à une valeur à l'aide de la clé, soit à la manière d'un tableau soit avec un point :

```
console.log(object['key2']); // "Value"
console.log(object.key2); // "Value"
```

JavaScript

Les objets

Il est possible de créer une clé directement en l'assignant

```
object['newkey'] = "newValue";
object.otherNewKey = "OtherNewValue";
```

Il est possible de supprimer une clé avec l'instruction `delete`

```
delete object['newkey'];
delete object.otherNewKey;
```

JavaScript

Traverser des tableaux et des objets

Il existe la boucle `for(... of ...){}` pour itérer dans des tableau :

```
const table = ["a", "c", "e", "g"];
for(const value of table){
    console.log(value) // a c e g
}
```

Il existe la boucle `for(... in ...){}` pour itérer dans des objets :

```
const obj = {
    'k1' : "v1",
    'k2' : 3,
    'k3' : true
};
for(const key in obj){
    console.log(key + " : " + obj[key]); // k1 : v1    k2 : 3    k3 : true
}
```

Vous ne pouvez pas modifier le contenu du tableau / objet avec ces boucles. Modifier "value" ou "key" (s'ils ne sont pas const) ne changera pas l'objet itéré.

A noter que dans ces deux types de boucles, l'itérateur peut être déclarer avec `const` (`const value` et `const key`). Cela est possible car ces boucles créent une nouvelle variable à chaque itération. Le `const` assure ici que la valeur n'est pas modifiée durant une itération.

Immuabilité

JavaScript

L'immuabilité

En JavaScript, il existe une différence fondamentale entre les types de base et les objets.

Qu'affiche ce code ?

```
let a = 5;
let b = a;

b = 10;

console.log(a);
console.log(b);
```

Voici un exemple simple, rien de surprenant, mais il est intéressant d'examiner en détail ce que JavaScript fait lors de l'execution de ces lignes.

JavaScript

L'immuabilité

En JavaScript, il existe une différence fondamentale entre les types de base et les objets.

Qu'affiche ce code ?

```
let a = 5;
let b = a;

b = 10;

console.log(a);
console.log(b);
```

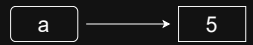
Ce code affichera bien :

```
5
10
```

Voici un exemple simple, rien de surprenant, mais il est intéressant d'examiner en détail ce que JavaScript fait lors de l'execution de ces lignes.

JavaScript

L'immuabilité



Les types de base sont des variables qui *ne changent jamais de valeur*. On dit qu'elles sont immuables. Lors d'un assignment, JavaScript va copier la valeur dans un nouvel espace mémoire, indépendant du premier. Ainsi, modifier la variable `b` dans l'exemple précédent ne modifie pas `a`, car il s'agit de deux variables séparées.

En réalité, si vous assignez une nouvelle valeur à `a`, par exemple avec le code :

```
let a = 5;
a = 10;
```

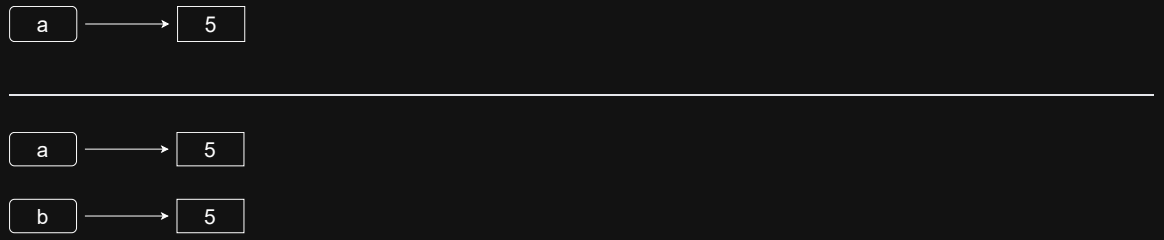
La première ligne assignera à `a` une valeur 5 stockée en mémoire, puis la seconde ligne créera un autre espace mémoire et assignera ce dernier à `a`. La valeur d'origine `5` ne sera pas modifiée. On peut le constater si `a` est une chaîne de caractères :

```
let a = "Bonjour";
a[0] = "A";
```

Faire un `console.log(a)` affichera "Bonjour" et non "Aonjour", même si on a assigné "A" au premier caractère. Cet assignment n'a eu aucun effet car `a` est d'un type de base et donc immuable.

JavaScript

L'immuabilité



Les types de base sont des variables qui *ne changent jamais de valeur*. On dit qu'elles sont immuables. Lors d'un assignment, JavaScript va copier la valeur dans un nouvel espace mémoire, indépendant du premier. Ainsi, modifier la variable `b` dans l'exemple précédent ne modifie pas `a`, car il s'agit de deux variables séparées.

En réalité, si vous assignez une nouvelle valeur à `a`, par exemple avec le code :

```
let a = 5;
a = 10;
```

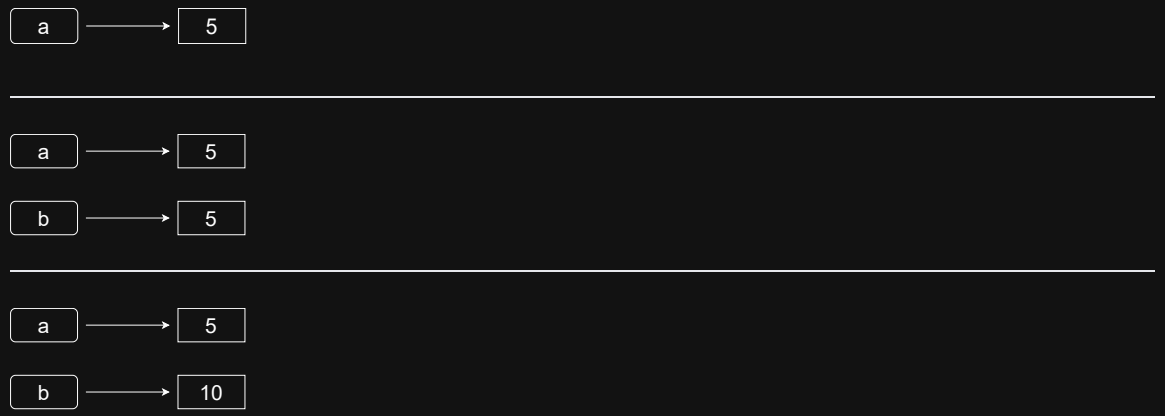
La première ligne assignera à `a` une valeur 5 stockée en mémoire, puis la seconde ligne créera un autre espace mémoire et assignera ce dernier à `a`. La valeur d'origine `5` ne sera pas modifiée. On peut le constater si `a` est une chaîne de caractères :

```
let a = "Bonjour";
a[0] = "A";
```

Faire un `console.log(a)` affichera "Bonjour" et non "Aonjour", même si on a assigné "A" au premier caractère. Cet assignment n'a eu aucun effet car `a` est d'un type de base et donc immuable.

JavaScript

L'immuabilité



Les types de base sont des variables qui *ne changent jamais de valeur*. On dit qu'elles sont immuables. Lors d'un assignment, JavaScript va copier la valeur dans un nouvel espace mémoire, indépendant du premier. Ainsi, modifier la variable `b` dans l'exemple précédent ne modifie pas `a`, car il s'agit de deux variables séparées.

En réalité, si vous assignez une nouvelle valeur à `a`, par exemple avec le code :

```
let a = 5;
a = 10;
```

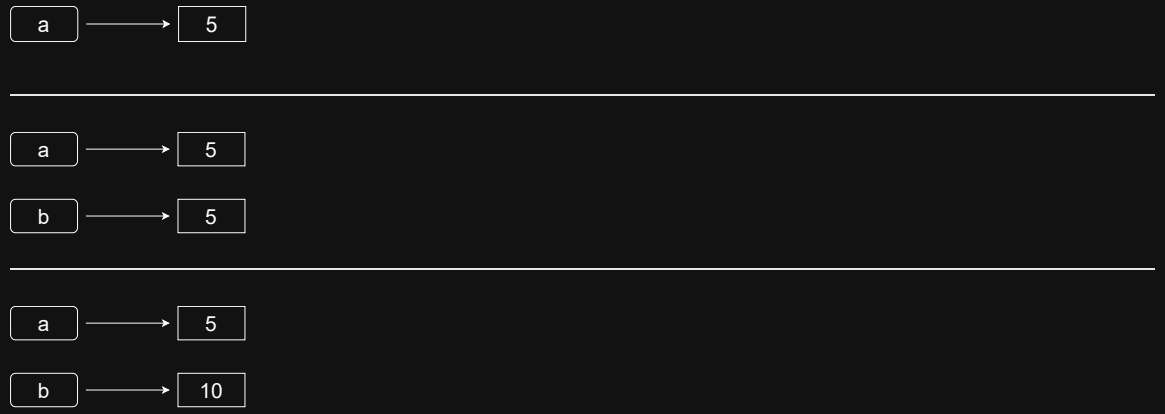
La première ligne assignera à `a` une valeur 5 stockée en mémoire, puis la seconde ligne créera un autre espace mémoire et assignera ce dernier à `a`. La valeur d'origine `5` ne sera pas modifiée. On peut le constater si `a` est une chaîne de caractères :

```
let a = "Bonjour";
a[0] = "A";
```

Faire un `console.log(a)` affichera "Bonjour" et non "Aonjour", même si on a assigné "A" au premier caractère. Cet assignment n'a eu aucun effet car `a` est d'un type de base et donc immuable.

JavaScript

L'immuabilité



En JavaScript les types de base sont *immuables*. Un assignment créera toujours une copie.

Les types de base sont des variables qui *ne changent jamais de valeur*. On dit qu'elles sont immuables. Lors d'un assignment, JavaScript va copier la valeur dans un nouvel espace mémoire, indépendant du premier. Ainsi, modifier la variable `b` dans l'exemple précédent ne modifie pas `a`, car il s'agit de deux variables séparées.

En réalité, si vous assignez une nouvelle valeur à `a`, par exemple avec le code :

```
let a = 5;
a = 10;
```

La première ligne assignera à `a` une valeur 5 stockée en mémoire, puis la seconde ligne créera un autre espace mémoire et assignera ce dernier à `a`. La valeur d'origine `5` ne sera pas modifiée. On peut le constater si `a` est une chaîne de caractères :

```
let a = "Bonjour";
a[0] = "A";
```

Faire un `console.log(a)` affichera "Bonjour" et non "Aonjour", même si on a assigné "A" au premier caractère. Cet assignment n'a eu aucun effet car `a` est d'un type de base et donc immuable.

JavaScript

L'immuabilité

En JavaScript, il existe une différence fondamentale entre les types de bases et les autres objets.

Et qu'affiche ce code ?

```
let a = [1, 2, 3];
let b = a;

b[1] = 10;

console.log(a);
console.log(b);
```

Ici on remarque que modifier `b` modifie également `a`. Le comportement est différent que celui du cas précédent, pourquoi?

JavaScript

L'immuabilité

En JavaScript, il existe une différence fondamentale entre les types de bases et les autres objets.

Et qu'affiche ce code ?

```
let a = [1, 2, 3];
let b = a;

b[1] = 10;

console.log(a);
console.log(b);
```

Ce code affichera :

```
[1, 10, 3]
[1, 10, 3]
```

Ici on remarque que modifier `b` modifie également `a` . Le comportement est différent que celui du cas précédent, pourquoi?

JavaScript

L'immuabilité

Contrairement aux types de base, quand on assigne un objet, celui-ci n'est pas copié. La variable assignée pointera vers le même espace mémoire que la variable originale. Ainsi, changer l'une changera implicitement l'autre aussi.

Les objets sont *mutable*, il est possible de les changer directement sans créer de copie. C'est bien le comportement attendu :

```
let a = [1,2,3];
a[0] = 4;
```

Dans ce cas un `console.log(a)` affichera bien `[4,2,3]`, car le tableau a été modifié.

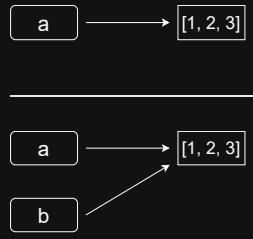
Il devient clair pourquoi un code comme celui-ci est valide :

```
const a = [1,2,3];
a[0] = 10;
```

Bien que `a` soit `const`, il est autorisé de modifier un élément du tableau, car `a` est toujours assigné au même objet. C'est l'objet lui-même que l'on change avec la seconde instruction. Le mot clé `const` ne garanti donc *pas* que l'objet assigné à une variable ne changera pas, il garanti que la variable ne pourra pas être assignée à autre chose.

JavaScript

L'immuabilité



Contrairement aux types de base, quand on assigne un objet, celui-ci n'est pas copié. La variable assignée pointera vers le même espace mémoire que la variable originale. Ainsi, changer l'une changera implicitement l'autre aussi.

Les objets sont *mutable*, il est possible de les changer directement sans créer de copie. C'est bien le comportement attendu :

```
let a = [1,2,3];
a[0] = 4;
```

Dans ce cas un `console.log(a)` affichera bien `[4,2,3]`, car le tableau a été modifié.

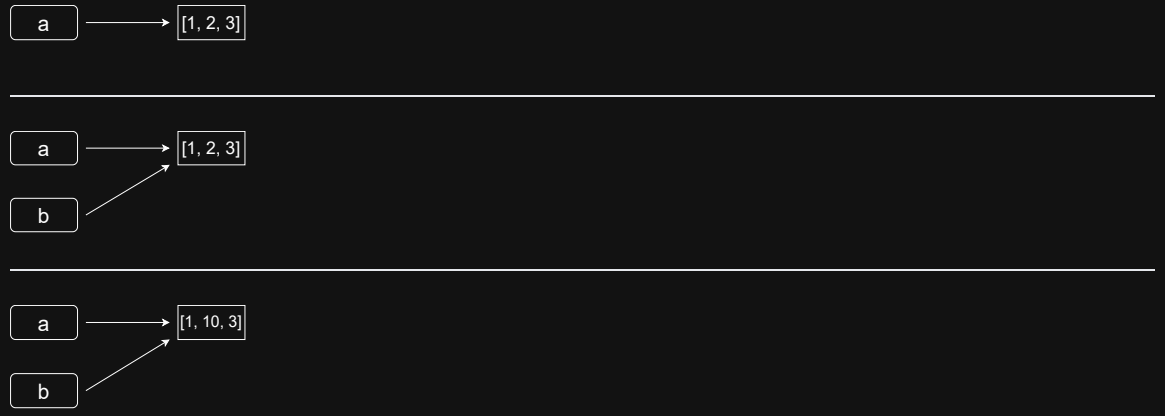
Il devient clair pourquoi un code comme celui-ci est valide :

```
const a = [1,2,3];
a[0] = 10;
```

Bien que `a` soit `const`, il est autorisé de modifier un élément du tableau, car `a` est toujours assigné au même objet. C'est l'objet lui-même que l'on change avec la seconde instruction. Le mot clé `const` ne garanti donc *pas* que l'objet assigné à une variable ne changera pas, il garanti que la variable ne pourra pas être assignée à autre chose.

JavaScript

L'immuabilité



Contrairement aux types de base, quand on assigne un objet, celui-ci n'est pas copié. La variable assignée pointera vers le même espace mémoire que la variable originale. Ainsi, changer l'une changera implicitement l'autre aussi.

Les objets sont *mutable*, il est possible de les changer directement sans créer de copie. C'est bien le comportement attendu :

```
let a = [1,2,3];
a[0] = 4;
```

Dans ce cas un `console.log(a)` affichera bien `[4,2,3]`, car le tableau a été modifié.

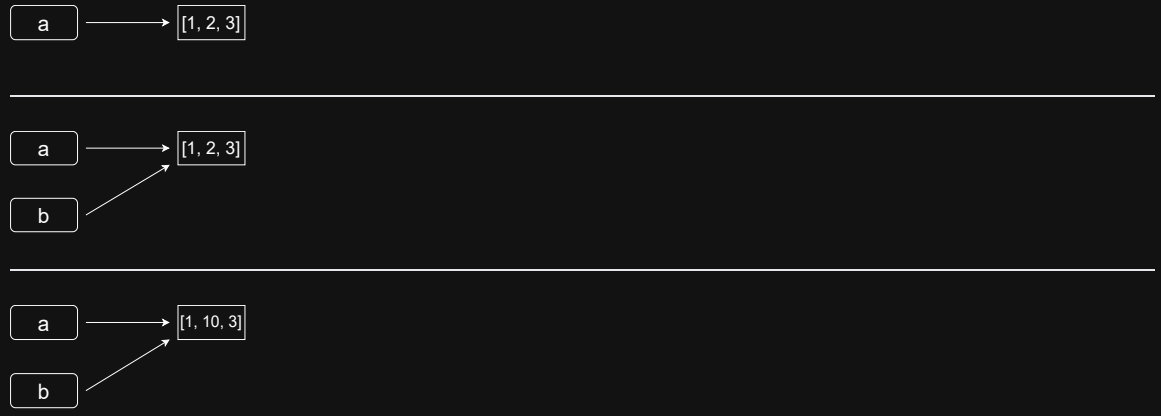
Il devient clair pourquoi un code comme celui-ci est valide :

```
const a = [1,2,3];
a[0] = 10;
```

Bien que `a` soit `const`, il est autorisé de modifier un élément du tableau, car `a` est toujours assigné au même objet. C'est l'objet lui-même que l'on change avec la seconde instruction. Le mot clé `const` ne garanti donc *pas* que l'objet assigné à une variable ne changera pas, il garanti que la variable ne pourra pas être assignée à autre chose.

JavaScript

L'immuabilité



En JavaScript les types non basiques sont *mutables*. Un assignment créera une référence vers l'objet.

Contrairement aux types de base, quand on assigne un objet, celui-ci n'est pas copié. La variable assignée pointera vers le même espace mémoire que la variable originale. Ainsi, changer l'une changera implicitement l'autre aussi.

Les objets sont *mutable*, il est possible de les changer directement sans créer de copie. C'est bien le comportement attendu :

```
let a = [1,2,3];
a[0] = 4;
```

Dans ce cas un `console.log(a)` affichera bien `[4,2,3]`, car le tableau a été modifié.

Il devient clair pourquoi un code comme celui-ci est valide :

```
const a = [1,2,3];
a[0] = 10;
```

Bien que `a` soit `const`, il est autorisé de modifier un élément du tableau, car `a` est toujours assigné au même objet. C'est l'objet lui-même que l'on change avec la seconde instruction. Le mot clé `const` ne garanti donc *pas* que l'objet assigné à une variable ne changera pas, il garanti que la variable ne pourra pas être assignée à autre chose.

Les fonctions

JavaScript

Les fonctions

Les fonctions se déclarent de la manière suivante

```
function add(a,b,c){    // Déclare la fonction "add", disponible dans le contexte courant
    return a + b + c;
}

const result = add(4,6,8); // result = 18
console.log(typeof add); // Affiche "function"
```

En Javascript les fonctions sont des objets comme les autres, il est possible de les assigner à des variables :

```
const maFonction = function foo(a, b) { // Déclare une fonction "foo" assignée à la variable "maFonction"
    return a * b;
}
console.log(maFonction(3,4)) // Affiche 12
console.log(foo(3,4)) // ERREUR : foo n'est pas définit
```

Dans cet exemple, on a nommé notre fonction "foo". Ce nom est inutile car notre fonction est stockée dans "maFonction".

Les fonctions en JavScript se déclarent avec une syntaxe similaire au C/C++. Dans le premier exemple on déclare une fonction "add" qui prends 3 paramètres. Le corps de la fonction retourne l'addition des trois paramètres.
Comme dans beaucoup de languages, l'instruction "return" retourne la valeur indiquée juste après et met fin à l'execution de la fonction.
En JavaScript les fonctions sont considérées comme des objets, qui peuvent donc être assignés à des variables. Il est possible de créer des tableaux de fonctions, des objets contenant des fonctions, etc. Quand une fonction est assignée à une variable, elle n'est pas disponible dans le contexte actuel. Elle est simplement "stockée" dans la variable

JavaScript

Les fonctions

Stocker des fonctions dans des variables est très courant en JavaScript. Il existe une syntaxe alternative sans donner de nom à une fonction. On appelle cela une fonction anonyme.

```
const operation1 = function nom_inutile(a,b){ return 1 + a * b } // Syntaxe classique
const operation2 = function (a,b){ return 1 + a * b } // Fonction anonyme
```

Les fonctions anonymes sont très utilisées en JavaScript, il existe une notation alternative, plus légère

```
const operation3 = (a,b) => { return 1 + a * b;} // Syntaxe plus légère, avec une fonction anonyme
```

Si la fonction n'est composée que d'un `return` (comme c'est le cas ici), il est même possible de l'enlever ainsi que les accolades.

```
const operation4 = (a,b) => 1 + a * b; // Dans ce cas, on peut omettre les accolades et le return
```

Toutes les notations de cette slide sont équivalentes

On le verra plus loins, mais les fonctions et leur gestion est au coeur du JavaScript. Il est extrêmement commun d'avoir des fonctions dans des objets, ou en tant que paramètres d'autres fonctions, etc.

JavaScript

Les fonctions

Exemple où une fonction est passée en paramètre d'une fonction :

```
function apply_to_table(table, func){       // Le paramètre func est une fonction appliquée au tableau
    for(let i=0; i<table.length; i++){      // On itère pour chaque élément du tableau
        table[i] = func(table[i]);          // Appel de la fonction avec l'élément du tableau en paramètre
    }
    return table;
}
console.log(apply_to_table([2, 4, 6], (x) => 3*x+1)); // Affiche [7, 13, 19]
```

Cet exemple illustre la puissance des fonctions en JavaScript : elles peuvent être passées en tant que valeur n'importe où. Du point de vue de JavaScript, les fonctions sont des objets, qui ont la particularité de contenir du code pouvant être executé.

JavaScript

Les fonctions – Paramètres par défaut

Les fonctions acceptent des paramètres par défaut.

Exemple :

```
function add(a, b, c=0, d=0){
    return a + b + c + d;
}

console.log(add(2,3));      // Affiche 5
console.log(add(2,3,4));    // Affiche 9
console.log(add(2,3,4,5));  // Affiche 14
```

Les paramètres par défaut doivent toujours se trouver en dernière position `function add(a=0, b){}` est invalide

JavaScript

Les fonctions – Immutabilité

L'immutabilité s'applique aussi dans le cas des fonctions, de la même manière que pour l'assignation.

```
function modify_number(num){    // Type de base en paramètre, il sera dupliqué durant l'execution
    num = 10;                   // Cette assignation modifie la copie local dans la fonction
    return num;
}

const num_1 = 1;
const num_2 = modify_number(num_1);
console.log(num_1);     // Retourne 1  : La valeur de base n'a pas été modifiée
console.log(num_2);     // Retourne 10 : Il s'agit de la valeur renvoyée par la fonction


function modify_table(table){   // Type composé en paramètre, ce sera une référence à l'objet de base
    table[1] = 10;              // Cette assignation va modifier le tableau original
    return table;
}

const table_1 = [1,2,3];
const table_2 = modify_table(table_1);
console.log(table_1);   // Retourne [10, 2, 3] : table_1 a été modifié par la fonction
console.log(table_2);   // Retourne [10, 2, 3] : il s'agit du même tableau que table_1
```

Quand une fonction est appelée, au moment d'assigner la valeur à ses paramètres, elle se comporte comme l'opérateur `=`. C'est-à-dire :

- Les types de base sont immuable, et seront donc dupliqués
- Les objets (types composés) sont mutables, et la fonction s'exécutera *sur le même objet que passé en paramètre*

La fonction *modify_number* ci-dessus prend un nombre en paramètre, et lui assigne 10. Comme il s'agit d'un nombre, il sera copié au moment d'executer la fonction, cela ne modifiera pas le nombre d'origine, et la fonction retournera toujours 10.

La fonction *modify_table*, au contraire, prend un tableau en paramètre. Le tableau sur lequel va agir la fonction sera le tableau passé en paramètre : il n'y a pas de copie. La ligne `table[1] = 10` va donc modifier le tableau d'origine. De plus, la ligne `return table` va renvoyer le même tableau que passé en paramètre.

Les variables `table_1` et `table_2` vont donc référencer le même tableau, et tout changement de l'une sera visible dans l'autre.

Les fonctions qui modifient leur objets d'entrée sans créer de copie sont souvent appelée *in-place*.

JavaScript

Les fonctions – Immutabilité

L'immutabilité s'applique aussi dans le cas des fonctions, de la même manière que pour l'assignation.

```
function modify_number(num){    // Type de base en paramètre, il sera dupliqué durant l'execution
    num = 10;                   // Cette assignation modifie la copie local dans la fonction
    return num;
}

const num_1 = 1;
const num_2 = modify_number(num_1);
console.log(num_1);     // Retourne 1  : La valeur de base n'a pas été modifiée
console.log(num_2);     // Retourne 10 : Il s'agit de la valeur renvoyée par la fonction


function modify_table(table){   // Type composé en paramètre, ce sera une référence à l'objet de base
    table[1] = 10;              // Cette assignation va modifier le tableau original
    return table;
}

const table_1 = [1,2,3];
const table_2 = modify_table(table_1);
console.log(table_1);   // Retourne [10, 2, 3] : table_1 a été modifié par la fonction
console.log(table_2);   // Retourne [10, 2, 3] : il s'agit du même tableau que table_1
```

Quand une fonction est appelée, au moment d'assigner la valeur à ses paramètres, elle se comporte comme l'opérateur `=`. C'est-à-dire :

- Les types de base sont immuable, et seront donc dupliqués
- Les objets (types composés) sont mutables, et la fonction s'exécutera *sur le même objet que passé en paramètre*

La fonction *modify_number* ci-dessus prend un nombre en paramètre, et lui assigne 10. Comme il s'agit d'un nombre, il sera copié au moment d'executer la fonction, cela ne modifiera pas le nombre d'origine, et la fonction retournera toujours 10.

La fonction *modify_table*, au contraire, prend un tableau en paramètre. Le tableau sur lequel va agir la fonction sera le tableau passé en paramètre : il n'y a pas de copie. La ligne `table[1] = 10` va donc modifier le tableau d'origine. De plus, la ligne `return table` va renvoyer le même tableau que passé en paramètre.

Les variables `table_1` et `table_2` vont donc référencer le même tableau, et tout changement de l'une sera visible dans l'autre.

Les fonctions qui modifient leur objets d'entrée sans créer de copie sont souvent appelée *in-place*.

Le DOM

JavaScript

Le DOM

Comment lire et écrire le contenu de notre document HTML avec du JavaScript ? Grâce au Document Object Model (DOM) !

Le DOM est l'interface permettant d'accéder à la page web. Il se caractérise par l'ajout de deux objets, accessibles partout dans le code : `document` et `window`.

Pour récupérer des éléments HTML :

```
const e1 = document.getElementById('monId');           // Retourne l'élément qui porte l'id "monId"
const e2 = document.getElementsByClassName('maClasse'); // Retourne un tableau avec les éléments de classe "maClasse"
const e3 = document.querySelector('p .large');          // Retourne le premier élément correspondant au sélecteur CSS
```

Jusqu'ici, nous n'avons utilisé que la console pour interagir avec JavaScript. Évidemment cela n'est pas très pratique et la raison principale d'executer du JavaScript dans un navigateur est de pouvoir interagir avec la page web et son contenu.

Le DOM est l'interface que met à disposition le navigateur pour lire / écrire du contenu sur la page.
L'objet `document` offre des fonctions pour interagir avec la page web elle-même et son contenu.
L'objet `window` offre des fonctions pour interagir avec la fenêtre du navigateur. Par exemple, récupérer la résolution de la fenêtre.

JavaScript

Le DOM

Une fois un élément récupéré, toutes ses propriétés sont modifiables

```
const elem = document.querySelector('p');           // Sélectionne le premier paragraphe de la page
elem.textContent = "Nouveau contenu du paragraphe"; // Assigne un nouveau texte au paragraphe
elem.style.fontSize = "20pt";                       // Assigne une nouvelle taille de police
```

Référence du DOM et ses fonctions : https://www.w3schools.com/js/js_htmldom.asp

La fonction `addEventListener` permet d'attacher une action à un événement.

Le premier paramètre est une chaîne de caractère qui décrit l'événement.

Le second paramètre est l'action à executer quand l'événement survient. Cette action est une fonction, c'est un excellent exemple d'une fonction qui prend en paramètre une autre fonction à executer plus tard.

JavaScript

Le DOM

Une fois un élément récupéré, toutes ses propriétés sont modifiables

```
const elem = document.querySelector('p');            // Sélectionne le premier paragraphe de la page
elem.textContent = "Nouveau contenu du paragraphe";  // Assigne un nouveau texte au paragraphe
elem.style.fontSize = "20pt";                        // Assigne une nouvelle taille de police
```

Référence du DOM et ses fonctions : https://www.w3schools.com/js/js_htmldom.asp

Il est aussi possible d'attacher des événements aux éléments :

```
const elem = document.getElementById('MyId');                  // Récupère l'élément à modifier
elem.addEventListener("click", ()=>elem.textContent="Clic!");   // Au clic, modifie le texte de l'élément à "Clic!"
```

```
// document.body retourne toujours la balise body de la page actuelle
// Quand la page est complètement chargée ("load"), change le texte de l'élément info en "Loaded!"
document.body.addEventListener("load", ()=>document.getElementById('info').textContent="Loaded!");
```

La fonction `addEventListener` permet d'attacher une action à un événement.

Le premier paramètre est une chaîne de caractère qui décrit l'événement.

Le second paramètre est l'action à executer quand l'événement survient. Cette action est une fonction, c'est un excellent exemple d'une fonction qui prend en paramètre une autre fonction à executer plus tard.

JavaScript

Le DOM - Les événements

Il est aussi possible de spécifier un événement directement dans la balise :

index.html

```
<!DOCTYPE html>
<html>
    <head>...</head>
    <body onload="execute_on_load()">
        <button onclick="execute_on_click()">Button</button>
        <script src="script.js"></script>
    </body>
</html>
```

script.js

```
function execute_on_load(){console.log("Loaded !");}

function execute_on_click(){console.log("Clicked !");}
```

Programmation orientée objet

JavaScript

Les objets et les classes

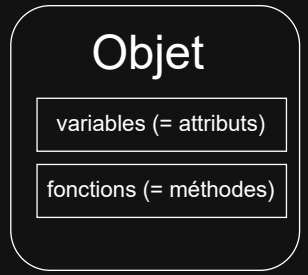
La programmation orientée objet est une manière de concevoir du code (appelé paradigme) dont le principe est de découper son code en objets. Un objet peut représenter un élément visible à l'écran, un concept concret ou abstrait, ou n'importe quelle "unité" qui possède une logique interne.

Un objet contient deux catégories d'éléments :

- Ses variables internes (nommées attributs)
- Ses fonctions internes (nommées méthodes)

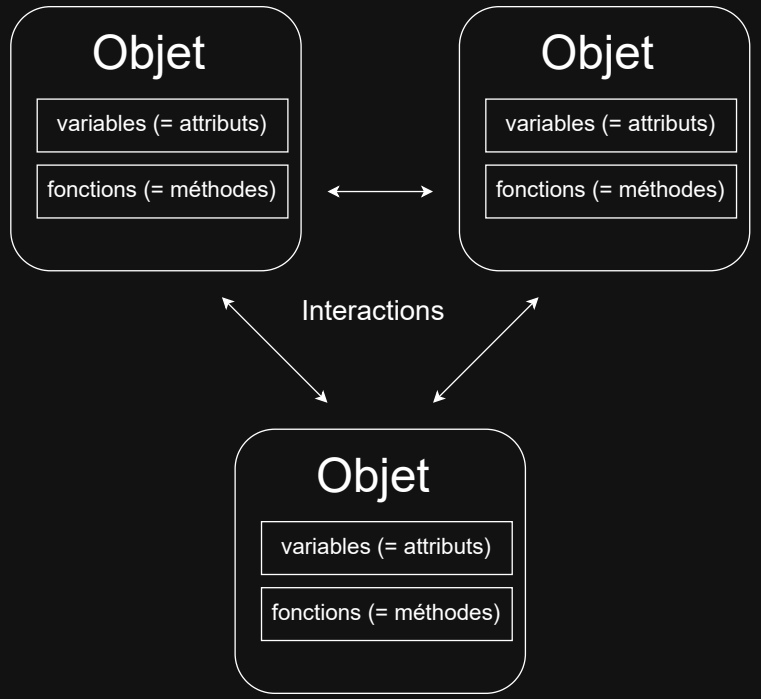
JavaScript

Les objets et les classes



JavaScript

Les objets et les classes



JavaScript

Les objets et les classes

Exemple

```
const rectangle = {
    'height' : 300,                                     // Attribut "height"
    'width' : 200,                                      // Attribut "width"
    'area' : function () { return this.height * this.width }    // Méthode "area"
}
```

L'opérateur `this` permet d'obtenir l'objet actuel

On pourrait définir un objet avec le code suivant. Mais cela est peu pratique.

- Il faudrait déclarer chaque objet séparément, ou alors créer une fonction spécifique pour créer des rectangles
- Pour des objets plus complexe, la syntaxe devient rapidement lourde et difficile à lire

JavaScript

Les objets et les classes

Exemple

```
const rectangle = {
    'height' : 300,                                     // Attribut "height"
    'width' : 200,                                      // Attribut "width"
    'area' : function () { return this.height * this.width }    // Méthode "area"
}
```

L'opérateur `this` permet d'obtenir l'objet actuel

```
rectangle.width         // Retourne 200
rectangle.area()        // Retourne 60000
rectangle.width = 5
rectangle.area()        // Retourne 1500
```

Que faire si on veut créer plusieurs rectangles ? Il faudra à chaque fois tout redéclarer...

On pourrait définir un objet avec le code suivant. Mais cela est peu pratique.

- Il faudrait déclarer chaque objet séparément, ou alors créer une fonction spécifique pour créer des rectangles
- Pour des objets plus complexe, la syntaxe devient rapidement lourde et difficile à lire

JavaScript

Les objets et les classes

Il existe une syntaxe spécialement conçue pour les objets. Une classe est la "description" d'un objet, permettant facilement d'instancier plusieurs objets par la suite.

```
class Rectangle{
    constructor(width, height){
        this.width = width;
        this.height = height;
    }
    area(){
        return this.width * this.height;
    }
}
```

Avec le mot clé `new` il est possible d'instancier des classes facilement

```
const rec1 = new Rectangle(20, 30);
const rec2 = new Rectangle(200, 300);
rec1.area(); // 600
rec2.area(); // 60000
```

En utilisant les classes en javascript, il est possible d'avoir une syntaxe facile à lire et permettant d'instancier facilement des objets. La syntaxe utilise le mot clé "class" suivit du nom de la classe. Une classe est une description des propriétés d'un objet, une fois déclarée, une classe ne crée aucun objet. Il faudra utiliser le mot clé "new" pour instancier la classe en objet.
En résumé, une classe est la description d'un objet. Un objet est une instance d'une classe.

Une classe contient une méthode spéciale "constructor" qui s'execute lors de la création de l'objet. C'est elle qui reçoit les paramètres passés lors de l'instruction "new". Les autres méthodes se déclarent sans le mot clé "function", les unes après les autres dans la classe. Toutes les méthodes ont accès au "this" de l'objet courant.