

# JavaScript

---

# JavaScript

# Sommaire

1. Introduction au JavaScript
2. Les bases de JavaScript
3. Les variables
4. Type de données
5. Les opérateurs
6. Structures conditionnelles
7. Structures itératives
8. Les fonctions
9. Les tableaux
10. Les objets
11. Les classes
12. Promesses, Async Await

# Introduction au JavaScript

# Qu'est-ce que JavaScript ?

- JavaScript a initialement été créé pour rendre les pages *"dynamiques"*
- Les programmes dans ce langage sont appelés **scripts**
- Ils peuvent être écrits directement dans une page HTML et exécutés automatiquement au chargement des pages par les navigateurs
- ⚠ JavaScript et Java sont deux langages très différents

# Comment s'exécute JavaScript

- Les navigateurs intègrent des moteurs que l'on appelle également machine virtuelle JavaScript
  - **V8** : Chrome, Edge, Opera
  - **SpiderMonkey** : Firefox
  - **Chakra** : (feu) Internet Explorer
- Le JavaScript est un langage utilisé aussi bien côté serveur que côté navigateur

# Langage interprété

- En informatique, on parle de code **interprété** ou **compilé**
- JavaScript est un langage **interprété** : le code est exécuté de haut en bas et le résultat du code exécuté est envoyé immédiatement
- Les langages compilés quant à eux sont transformés (compilés) en une autre forme avant d'être exécutés par l'ordinateur (C, C++)

# Avantages de JavaScript

- Intégration complète avec HTML et CSS
- Syntaxe simple et claire
- Prise en charge par tous les navigateurs et activé par défaut



# Les bases de JavaScript

# La balise script

- Les programmes JavaScript peuvent être insérés dans n'importe quelle partie d'un document HTML à l'aide de la balise <script>
- La balise script contient du code qui est automatiquement exécuté lorsque le navigateur rencontre la balise

```
...  
<body>  
  <script>  
    alert( 'Hello, world!' );  
  </script>  
</body>  
...
```

# Scripts externes

- Il est généralement conseillé de séparer le JavaScript de la page HTML
- Les fichiers JavaScript ont une extension `.js`
- Pour l'importer il suffit d'ajouter l'attribut `src` à la balise script
- Les fichiers sont téléchargés puis stockés dans le cache

```
<script src="/chemin/vers/script.js"></script>
```

# Instruction

- Les instructions sont des constructions de syntaxe et des commandes qui effectuent des actions
- Chaque instruction est généralement écrite sur une ligne distincte
- Les instructions finissent par `;` même si celui-ci est *généralement* facultatif lors d'un retour à la ligne

```
alert("Hello");  
alert("World");
```

# Les commentaires

- Les commentaires sur une ligne commencent par deux barres obliques `//`

```
alert("World"); // Ceci est un commentaire
```

- Les commentaires multilignes commencent par une barre oblique et un astérisque `/*` et se termine par un astérisque et une barre oblique `*/`

```
/* Un exemple avec deux messages.  
C'est un commentaire multiligne.  
*/  
alert("Hello");
```

# Variables

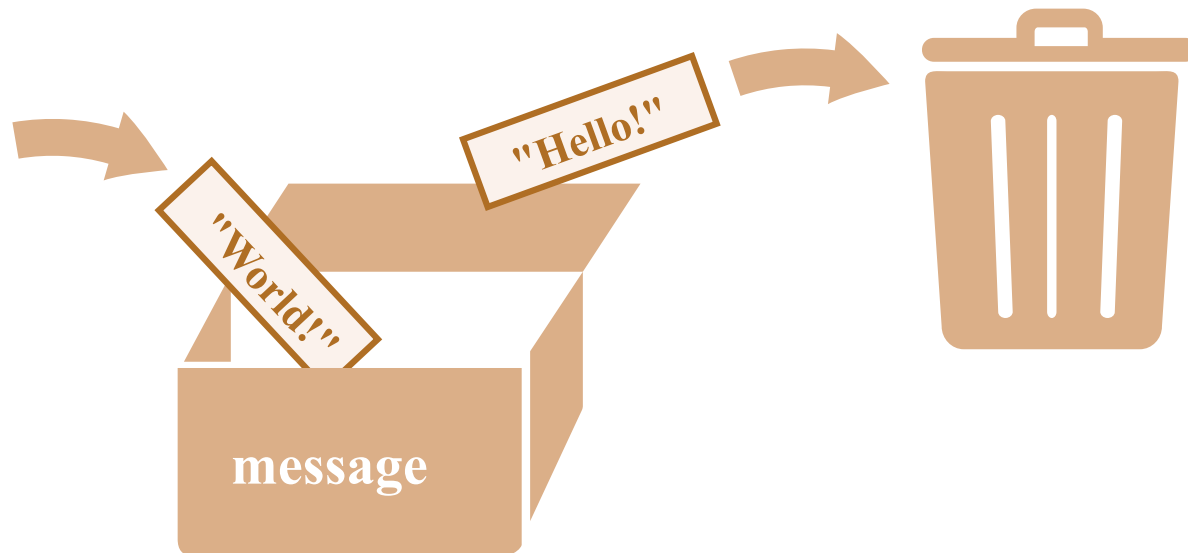
# Les variables

- Une variable est un stockage nommé pour les données
- Elle se comporte comme une boîte dans laquelle on stocke des informations
- Pour créer une variable en JavaScript, il suffit d'utiliser le mot-clé `let`

```
let message;  
  
message = "Hello"; // stocke la chaîne de caractères 'Hello'  
  
let prenom = "Toto"; // Déclaration et affectation
```

# Fonctionnement d'une variable

- Lorsque la valeur est modifiée, les anciennes données sont supprimées de la variable





# Nom de variables

1. Le nom ne doit contenir que des **lettres**, des **chiffres**, des symboles \$ et \_
2. Le premier caractère ne doit pas être un chiffre
3. Lorsque le nom contient plusieurs mots, le **camelCase** est couramment utilisé: `monTresLongNom`
4. Les noms des variables sont sensibles à la casse
5. Ne pas utiliser des mots réservés : **return**, **class**, **let** ...

# Les constantes

- Pour déclarer une constante (non changeante), on peut utiliser `const` plutôt que `let`
- Elles ne peuvent pas être réassignées. Une tentative de le faire provoquerait une erreur
- On nomme généralement les constantes en majuscule pour des alias de valeurs stockées en "dur" dans le code

```
const myBirthday = "18.04.1982";  
const COLOR_ORANGE = "#FF7F00";
```

# Règle de nommage

Un nom de variable doit avoir une signification claire et évidente, décrivant les données qu'elle stocke

- Utiliser des noms lisibles par des humains comme `userName` ou `shoppingCart`
- Eviter les abréviations ou des noms courts
- Faire en sorte que le nom soit le plus descriptif et concis possible
- S'accorder avec son équipe (et soi-même) sur les termes utilisés

# Types de données

# Les types de données JavaScript

- Il existe 8 types de données de base en JavaScript

Type	Description	Exemple
number	entier ou virgule flottante	<code>let n = 123;</code>
bigint	entiers de longueur arbitraire	
string	chaînes de caractères	<code>let str = "Hello";</code>
boolean	true/false	<code>let nameFieldChecked = true;</code>
null	valeurs inconnues	<code>let age = null;</code>
undefined	valeurs non attribuées	<code>let age; alert(age);</code> // affiche "undefined"
symbol	identifiants uniques	
object	structures de données complexes	

# Type number

- Le type number sert à la fois à des nombres entiers et à des nombres à virgule flottante
- Outre les nombres réguliers il existe des valeurs spécifiques: `Infinity`, `-Infinity` et `NaN`
- Les opérations mathématiques ne provoquent jamais d'erreurs en JavaScript

# Détail sur le type Number

- Nan : La propriété globale NaN est une valeur utilisée pour représenter une quantité qui n'est pas un nombre (Not a Number)
- La division par 0 renvoie Infinity : `1/0 => Infinity`
- Une division qui provoque une erreur de calcul renvoie NaN : `"toto"/2 => NaN`

# Type BigInt

- BigInt a récemment été ajouté au langage pour représenter des entiers de longueur arbitraire
- Une valeur BigInt est créé en ajoutant n à la fin d'un entier
- À l'heure actuelle, BigInt est pris en charge dans Firefox/Chrome/Edge/Safari, mais pas dans IE

```
const bigInt = 1234567890123456789012345678901234567890n;
```



# Chaînes de caractères

- Une chaîne de caractères en JavaScript doit être entre guillemets
- JavaScript accepte 3 types de guillemets
  1. Double quotes "Hello"
  2. Single quotes 'Hello'
  3. Backticks `Hello`
- Les backticks permettent d'intégrer des variables et des expressions dans une chaîne en les encapsulant dans `${...}`

```
`the result is ${1 + 2}`;
```

# Type boolean

- Le type booléen n'a que deux valeurs: `true` et `false`
- Les valeurs booléennes sont généralement utilisées pour les comparaisons ou structures conditionnelles

```
let nameFieldChecked = true;  
let isGreater = 4 > 1;
```

# La valeur "null"

- La valeur spéciale null n'appartient à aucun type vu précédemment
- En JavaScript, null n'est pas une "référence à un objet non existant" ou un "pointeur nul" comme dans d'autres langages
- C'est une valeur spéciale qui a le sens de "rien", "vide" ou "valeur inconnue"

```
let age = null;
```

# La valeur "undefined"

- La valeur spéciale **undefined** se distingue des autres. C'est un type à part entière, comme null
- La signification de **undefined** est “la valeur n'est pas attribuée”.
- Si une variable est déclarée mais non affectée, alors sa valeur est exactement **undefined**

```
let age;  
  
alert(age); // affiche "undefined"
```

# Objects et Symbols

- les **objets** servent à stocker des collections de données et des entités plus complexe
- Le type **symbol** est utilisé pour créer des identificateurs uniques pour les objets

# L'opérateur typeof

- L'opérateur **typeof** renvoie le type de l'argument
- Il est utile lorsqu'on souhaite traiter différemment les valeurs de différents types ou de faire une vérification rapide

```
typeof undefined; // "undefined"
```

```
typeof 0; // "number"
```

```
typeof 10n; // "bigint"
```

```
typeof Math; // "object"
```

# Les opérateurs

# Opérateur d'affectation

Nom	Opérateur	Signification
Affectation	$x = f()$	$x = f()$
Affectation après addition	$x += f()$	$x = x + f()$
Affectation après soustraction	$x -= f()$	$x = x - f()$
Affectation après multiplication	$x *= f()$	$x = x * f()$
Affectation après division	$x /= f()$	$x = x / f()$
Affectation du reste	$x \% = f()$	$x = x \% f()$



# Opérateur mathématiques

Opérateur	Exemples
Addition	$1 + 1$
Soustraction	$1 - 1$
Division	$1 / 1$
Multiplication	$1 * 1$
Reste	$4 \% 2$
Exponentiel	$4 ** 2$
Incrément	$x++$ ou $++x$
Décrément	$x--$ ou $--x$

# Opérateurs de comparaison

Opérateur	Exemples
Égalité	<code>3 == var1</code>
Inégalité	<code>var1 != 4</code>
Égalité stricte	<code>3 === var1</code>
Inégalité stricte	<code>var1 !== "3"</code>
Supériorité stricte	<code>var2 &gt; var1</code>
Supériorité	<code>var2 &gt;= var1</code>
Infériorité stricte	<code>var1 &lt; var2</code>
Infériorité	<code>var1 &lt;= var2</code>

# Opérateur logiques

Opérateur	Utilisation	Description
ET logique (&&)	expr1 && expr2	renvoie true si les deux opérandes valent true sinon false
OU logique (  )	expr1    expr2	renvoie true si l'un des deux opérandes vaut true et false si les deux valent false
NON logique (!)	!expr	Renvoie false si son unique opérande peut être converti en true sinon renvoie true

# Les tableaux

# Définition

- Les tableaux sont des objets semblables à des **listes** dont le prototype possède des **méthodes** qui permettent de **parcourir** et de **modifier** le tableau
- L'objet global **Array** est utilisé pour créer des tableaux
- Les tableaux sont des objets de haut-niveau (en termes de complexité homme-machine)

# Déclaration d'un tableau

- Il existe deux syntaxes pour créer un tableau vide :

```
let fruits = [];  
// OU  
let fruits = Array();
```

- La première syntaxe est à privilégier

```
let fruits = ["Apple", "Banana"];
```

# Manipuler un tableau

- Accéder (via son index) à un élément du tableau

```
let first = fruits[0];
```

- Ajouter à la fin du tableau

```
let newLength = fruits.push("Orange");
```

- Supprimer le dernier élément du tableau

```
let last = fruits.pop();
```

- Supprimer le premier élément du tableau

```
let first = fruits.shift();
```

# Manipuler un tableau

- Ajouter au début du tableau

```
let newLength = fruits.unshift("Strawberry");
```

- Copier un tableau

```
let shallowCopy = fruits.slice();  
let copy = [...fruits];
```

- Parcourir un tableau `.forEach`

```
fruits.forEach((item, index) => console.log(item));
```



# Structures conditionnelles

# if

- L'instruction **if** exécute une instruction si une condition donnée est vraie ou équivalente à vrai
- Condition : Une expression qui est évaluée à **true** ou **false**

```
if (condition) {  
    statement1;  
}
```

## if ... else

- Si la clause else existe, l'instruction qui est exécutée si la condition est évaluée à false

```
if (condition) {  
    statement1;  
} else {  
    statement2;  
}
```

# if ... else if ... else

- Plusieurs instructions if...else peuvent être imbriquées afin de créer une structure else if

```
if (condition1)
    instruction1
else if (condition2)
    instruction2
else if (condition3)
    instruction3
...
else
    instructionN
```

# switch

- L'instruction `switch` évalue une **expression** et selon le résultat obtenu et le cas associé, exécute les instructions correspondantes

```
const code = 200;
switch (code) {
  case 200:
    console.log("Ok");
    break;
  ...
  default:
    console.log(`Code inconnu`);
}
```

# Opérateur ternaire

- L'opérateur ternaire **?** est fréquemment utilisé comme raccourci pour la déclaration des instructions : if...else
- `expression ? valeurSiVrai : valeurSiFaux;`

```
let elvisLives = Math.PI > 4 ? "Yep" : "Nope";
```

# Opérateur de coalescence

- Opérateur logique `??` qui renvoie son opérande de droite lorsque son opérande de gauche vaut null ou undefined
- `leftExpr ?? rightExpr`

```
const valA = valeurNulle ?? "valeur par défaut";
```

# Structures itératives



# While

- L'instruction while permet de créer une boucle qui s'exécute tant qu'une condition de test est vérifiée
- La condition est évaluée avant d'exécuter l'instruction contenue dans la boucle

```
let n = 0;

while (n < 3) {
  n++;
}

console.log(n);
```

# Do While

- L'instruction `do...while` crée une boucle qui exécute une instruction jusqu'à ce qu'une condition de test ne soit plus vérifiée
- le bloc d'instructions défini dans la boucle est donc exécuté au moins une fois

```
let i = 0;  
do {  
  console.log(i);  
  i++;  
} while (i < 5);
```

# For

- L'instruction for crée une boucle composée de trois expressions optionnelles séparées par des points-virgules et encadrées entre des parenthèses qui sont suivies par une instruction à exécuter dans la boucle

```
for (begin; condition; step) {  
    // Instructions  
}
```

```
for (let i = 0; i < 3; i++) {  
    console.log(i);  
}
```

# Continue et Break

- L'instruction **continue** arrête l'exécution des instructions pour l'itération de la boucle courante. L'exécution est reprise à l'itération suivante
- L'instruction **break** permet de terminer la boucle en cours et de passer le contrôle du programme à l'instruction suivant l'instruction terminée

# Les fonctions

# Définition

- une fonction est un « *sous-programme* » qui peut être appelé par du code extérieur à la fonction
- En JavaScript, les fonctions sont des **objets** de première classe
- Elles peuvent être manipulées et échangées, qu'elles peuvent avoir des **propriétés** et des **méthodes**, comme tous les autres objets JavaScript

# Déclaration de fonction

- Pour créer une fonction, nous pouvons utiliser une déclaration de fonction

```
function name(parameter1, parameter2, ...parameterN) {  
    // instructions  
}
```

```
function showMessage() {  
    alert("Hello everyone!");  
}
```

- Pour appeler une fonction : `showMessage();`

# Valeur par défaut

- Nous pouvons spécifier la valeur dite “par défaut” (à utiliser si omise) pour un paramètre dans la déclaration de fonction, en utilisant `=`

```
function showMessage(from, text = "Aucun texte") {  
    alert(from + ": " + text);  
}
```

```
showMessage("Arthur"); // Arthur: Aucun texte
```



# Renvoyer une valeur

- Une fonction peut renvoyer une valeur dans le code appelant en tant que résultat à l'aide du mot clé `return`
- Lorsque l'instruction `return` est rencontrée, la fonction se termine

```
function sum(a, b) {  
    return a + b;  
}  
  
let result = sum(1, 2);  
alert(result); // 3
```

# Expression de fonction

- En JavaScript, une fonction n'est pas une “structure de langage magique”, mais un **type de valeur** particulier
- L'omission d'un nom est autorisée pour les fonctions expressions

```
let sayHi = function () {  
    alert("Hello");  
};
```

# Callback

- Une callback est une fonction passée dans une autre fonction en tant qu'argument, qui est ensuite invoquée à l'intérieur de la fonction externe

```
function salutation(name) {  
    alert("Bonjour " + name);  
}  
function processUserInput(callback) {  
    var name = prompt("Entrez votre nom.");  
    callback(name);  
}  
processUserInput(salutation);
```

# Les fonctions fléchées

- Les fonctions fléchées sont une syntaxe raccourcie des expressions de fonctions

```
([param] [, param]) => {  
    //instructions  
}  
(param1, param2, ...,) => expression  
param => expression
```

```
let sum = (a, b) => a + b;
```

# Les objets

# Définition

- Un objet est un ensemble de propriétés et une propriété est une association entre un nom (aussi appelé clé) et une valeur
- La valeur d'une propriété peut être une fonction, auquel cas la propriété peut être appelée « méthode »
- Les objets sont utilisés pour stocker des collections de données variées et d'entités plus complexes
- Un objet peut être créé avec des accolades `{...}`, avec une liste optionnelle de propriétés

# Instancier un objet

- Il existe 2 syntaxes pour instancier un objet:

```
let user = new Object(); // syntaxe "constructeur d'objet"  
let user = {}; // syntaxe "littéral objet"
```

# Propriétés

- Les propriétés d'un objet sont des **variables** tout ce qu'il y a de plus classiques, exception faite qu'elles sont **attachées à des objets**
- Les propriétés d'un objet représentent ses **caractéristiques** et on peut y accéder avec une notation utilisant le point « . »
- La notation des propriétés se fait en camelCase

```
nomObjet.nomPropriete;
```

```
let maVoiture = {  
  make: "Ford",  
};
```



# Propriétés

- Les objets sont parfois appelés « tableaux associatifs »
- chaque propriété est associée avec une chaîne de caractères qui permet d'y accéder

```
maVoiture["fabricant"] = "Ford";  
maVoiture["modèle"] = "Mustang";  
maVoiture["année"] = 1969;
```

# Vérifier l'existence d'une propriété

- En JavaScript il est possible d'accéder à n'importe quelle propriété
- La lecture d'une propriété non existante renvoie simplement `undefined`
- Il existe également un opérateur spécial `in` pour tester l'existence d'une clé

```
let user = { name: "John", age: 30 };  
console.log("age" in user); // true
```

# For In

- Les boucles `for...in` qui permettent de parcourir l'ensemble des propriétés énumérables d'un objet et de sa chaîne de prototypes

```
for (const property in object) {  
  console.log(`${property}: ${object[property]}`);  
}
```

# Les méthodes

- On peut déclarer les méthodes de 2 manières en JavaScript
- La syntaxe à privilégiée est généralement la plus courte

```
user = {  
  sayHi: function () {  
    alert("Hello");  
  },  
};
```

```
user = {  
  sayHi() {  
    // identique à "sayHi: function(){...}"  
    alert("Hello");  
  },  
};
```

# Le mot clé this

- Il est courant qu'une méthode d'objet ait besoin d'accéder aux informations stockées dans l'objet pour effectuer son travail
- Pour accéder à l'objet, une méthode peut utiliser le mot-clé `this`

```
let user = {  
  name: "John",  
  sayHi() {  
    // this fait référence à l'objet courant  
    console.log(this.name);  
  },  
};
```

# Spécificité de this

- En JavaScript, this est “libre”, sa valeur est évaluée au moment de l’appel et ne dépend pas de l’endroit où la méthode a été déclarée, mais plutôt de l’objet “avant le point”
- `this` dans une fonction fléchées correspond à la valeur du contexte englobant

# La fonction constructeur

- Les fonctions constructeur sont techniquement des fonctions habituelles
- Elles permettent de créer des objets similaires
- Il existe cependant deux conventions :
  1. Elles sont nommées avec une lettre majuscule en premier
  2. Elles ne devraient être exécutées qu'avec l'opérateur `new`


# Déclaration d'un constructeur

- Quand une fonction est exécutée avec new:
  1. Un nouvel objet vide est créé et affecté à this
  2. Le corps de la fonction est exécuté
  3. La valeur de this est retournée

```
function User(name) {  
  this.name = name;  
  this.isAdmin = false;  
}  
  
let user = new User("Jack");
```



# Ajouter une méthode au constructeur

- L'utilisation de fonctions de constructeur pour créer des objets offre une grande flexibilité
-  Cette méthode n'est pas optimisée, il faudrait l'ajouter dans le prototype de la fonction

```
function User(name) {  
  this.name = name;  
  this.sayHi = function () {  
    alert("My name is: " + this.name);  
  };  
}  
john.sayHi(); // My name is: John
```

# Chaînage optionnel

- Le chaînage optionnel `?.` est un moyen sécurisé d'accéder aux propriétés d'objet imbriquées
- Le chaînage optionnel `?.` arrête l'évaluation si la valeur avant `?.` est `undefined` ou `null` et renvoie `undefined`

```
let user = {}; // l'utilisateur n'a pas d'adresse
console.log(user?.address?.street); // undefined (pas d'erreur)
user.admin?.(); // Execute une fonction si elle existe
user?.[key]; // accéder à la propriété
```

# Conversion d'objet en primitif

- La conversion objet à primitive est appelée automatiquement par de nombreuses fonctions intégrées
- L'algorithme de conversion est :
  1. Appeler `obj[Symbol.toPrimitive](hint)` si la méthode existe
  2. Sinon, si l'indice est "string", essaie `obj.toString()` puis `obj.valueOf()`
  3. Sinon, si l'indice est "number" ou "default", essaie `obj.valueOf()` puis `obj.toString()`

# Méthode toString()

- En pratique, il suffit souvent d'implémenter uniquement `obj.toString()` comme méthode pour les conversions de chaînes de caractères

```
let obj = {  
  toString() {  
    return "2";  
  },  
};
```

# Les classes

# Définition

- Les classes JavaScript ont été introduites avec ECMAScript 2015 (ES6)
- Elles sont un « *sucré syntaxique* » par rapport à l'héritage **prototypal**
- En réalité, les classes sont juste des **fonctions spéciales**
- Ainsi, les classes sont définies de la même façon que les fonctions : par **déclaration**, ou par **expression**

# Déclaration de classes

- Pour utiliser une déclaration de classe simple, on utilisera le mot-clé `class`, suivi par le nom de la classe que l'on déclare

```
class Rectangle {  
    constructor(hauteur, largeur) {  
        this.hauteur = hauteur;  
        this.largeur = largeur;  
    }  
}
```

# Constructeur

- La méthode constructor est une **méthode spéciale** qui permet de créer et **d'initialiser les objets** créés avec une classe
- Il ne peut y avoir **qu'une seule méthode** avec le nom "constructor" pour une classe donnée
- Si la classe contient plusieurs occurrences d'une méthode constructor, cela provoquera une exception **SyntaxError**



# Getter/Setter

- Tout comme les objets littéraux, les classes peuvent inclure des accesseurs/mutateurs, des propriétés évaluées etc

```
class User {  
  constructor(name) {  
    // invoque l'accesseur (the setter)  
    this.name = name;  
  }  
  
  get name() {  
    return this._name;  
  }  
}
```

# Héritage

- Le mot-clé `extends`, utilisé dans les déclarations de classes permet de créer une classe qui hérite d'une autre classe
- `super` permet d'accéder au prototype de l'objet auquel la méthode est attachée

```
class Mage extends Hero {  
    constructor(name, level, spell) {  
        super(name, level);  
        this.spell = spell;  
    }  
}
```

# Méthode statique

- Le mot-clé `static` permet de définir une méthode statique pour une classe
- Les méthodes statiques sont appelées par rapport à la classe entière et non par rapport à une instance donnée

```
class Calculer {  
    static somme(a, b) {  
        return a + b;  
    }  
}  
  
console.log(Calculer.somme(1, 2));
```

# Encapsulation

- une proposition expérimentale permet de définir des variables privées dans une classe avec le préfixe **#**

```
class ClassWithPrivateField {  
    #privateField;  
    #privateMethod() {  
        return this.#privateField;  
    }  
}
```

# Promesses, Async Await

# Asynchrone

- JavaScript est **synchrone** par défaut, et est exécuté en **single thread**
- Le code ne peut pas créer de nouveaux threads et s'exécuter en parallèle
- Les ordinateurs sont **asynchrones** par conception
- Asynchrone signifie que les choses peuvent se produire **indépendamment du flux principal** du programme

# Callback

- Une callback est une fonction simple qui est **transmise comme valeur** à une autre fonction et qui ne sera exécutée que lorsque l'événement se produira (exemple: click)
- JavaScript dispose de fonctions de première classe, qui peuvent être affectées à des variables et transmises à d'autres fonctions (appelées **higher-order functions**)

# Utilisation de callbacks

- Les callbacks sont utilisés partout, pas seulement dans les événements DOM

```
window.addEventListener("load", () => {  
    // La fenêtre est chargée  
    // faites ce que vous voulez  
});  
  
setTimeout(() => {  
    // s'exécute après 2 secondes  
}, 2000);
```



# Gestion des erreurs dans les callbacks

- Une stratégie courante consiste à définir le premier paramètre comme objet d'erreur : **error-first callback**
- S'il n'y a pas d'erreur, l'objet est null

```
fs.readFile("/file.json", (err, data) => {  
  if (err) {  
    console.log(err); // erreur  
    return;  
  }  
  
  console.log(data); // traitement des données  
});
```

# Le problème des callbacks

- Chaque callback ajoute un niveau d'imbrication ce qui peut ajouter de la complexité
- C'est ce qu'on appelle les **callback hell**

```

window.addEventListener("load", () => {
  document.getElementById("button").addEventListener("click", () => {
    setTimeout(() => {
      items.forEach((item) => {
        // votre code ici
      });
    }, 2000);
  });
});

```

# Les promesses

À partir de l'ES6, JavaScript a introduit plusieurs fonctionnalités qui nous aident avec du code asynchrone qui n'implique pas l'utilisation de callbacks : **Promesses** (ES6) et **Async/Await** (ES2017)

- Une promesse est un objet qui représente une éventuelle **réussite** ou **échec** d'un opérateur asynchrone
- Une promesse est un **objet** retourné auquel on attache des callbacks, au lieu de passer des callbacks dans une fonction

# État d'une promesse

- Une promesse peut avoir 3 états:
  - Opération en cours (*pending*)
  - Opération terminée avec succès (*fulfilled*)
  - Opération terminée/stoppée après un échec (*rejected*)
- La majorité des opérations asynchrones JavaScript sont déjà implémentées en utilisant les promesses

# Déclaration d'une promesse

- le constructeur Promise de JavaScript nécessite une fonction qui prend deux arguments : **resolve** et **reject**

```
let maPromesse = new Promise((resolve, reject) => {  
  // Tâche asynchrone  
  // Appel de resolve() si la promesse est résolue  
  // Appel de reject() si elle est rejetée  
});
```

# Exemple de promesse

```
function loadScript(src) {  
  return new Promise((resolve, reject) => {  
    let script = document.createElement("script");  
    script.src = src;  
    document.head.append(script);  
    script.onload = () => resolve("Fichier " + src + " bien chargé");  
    script.onerror = () => reject(new Error("Echec de chargement de " + src));  
  });  
}  
  
const promesse1 = loadScript("boucle.js");  
const promesse2 = loadScript("script2.js");
```

# Chaînage de promesses

- Le gestionnaire `.then` crée une nouvelle promesse avec le résultat obtenue par la fonction `resolve`
- Lorsqu'un gestionnaire renvoie une valeur, cela devient le résultat de cette promesse

```
maPromesse
  .then((resultat1) => {
    // Gérer le cas où la promesse est remplie avec succès
  })
  .then((resultat2) => {
    // Gérer le résultat renvoyé par le then précédent
  });
```

# Gestion des erreurs

- Les chaînes de promesses sont excellentes pour la gestion des erreurs
- Lorsqu'une promesse est rejetée, le contrôle saute au gestionnaire de rejet `.catch` le plus proche
- Généralement un objet de type Error est renvoyé

```
fetch("https://no-such-server.blabla")  
  .then((response) => response.json())  
  .catch((err) => alert(err));
```



# Méthodes statiques de Promise

- Il y a 6 méthodes statiques de la classe Promise
  1. `Promise.all(promises)` ★ : attend que toutes les promesses se résolvent et retourne un tableau de leurs résultats
  2. `Promise.allSettled(promises)` : attend que toutes les promesses se règlent et retourne leurs résultats sous forme de tableau d'objets
  3. `Promise.race(promises)` : attend que la première promesse soit réglée, et son résultat/erreur devient le résultat

# Async

- Il existe une syntaxe spéciale pour travailler avec les promesses d'une manière plus confortable, appelée “**async/await**”
- Le mot **async** devant une fonction signifie qu'elle renvoie toujours une promesse
- **async** s'assure que la fonction renvoie une promesse, et enveloppe les non-promesses dans celle-ci

```
async function f() {  
  return 1;  
}  
f().then(alert); // 1
```

# Await

- Le mot-clé `await` fait en sorte que JavaScript attende qu'une promesse se réalise et renvoie son résultat
- `await` est une syntaxe plus élégante pour obtenir le résultat de la promesse
- ⚠ `await` ne fonctionne que dans les fonctions asynchrones

```
async function showAvatar() {
  let response = await fetch("/article/promise-chaining/user.json");
  let user = await response.json();
  // ...
}
```

**Merci pour votre attention**

**Des questions ?**

