Javascript

m2iformation.fr

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Javascript

Sommaire

- Prérequis
- Debug VScode
- Rappels
  - fonctions de premier ordre
  - La portée des données
  - Pièges classiques du langage
  - POO en JavaScript
- Programmation fonctionnelle
- Programmation asynchrone
- Structuration et qualité du code
- API HTML5 en JavaScript
- JavaScript et Node.js
- Evolution ES7 à ES14

Prérequis

Prérequis

Ce support s'adresse à des développeurs ayant déjà une bonne compréhension des fondamentaux de JavaScript. Pour en tirer pleinement parti, il est indispensable de maîtriser les concepts de base tels que les types de données, les variables, les fonctions, les boucles, les conditions, les objets, ainsi que la manipulation du DOM. Une familiarité avec l'utilisation d'outils de développement dans un navigateur, l'écriture de scripts simples, et la gestion des événements est également attendue. Si ces notions ne sont pas encore acquises, il est fortement recommandé de suivre d'abord un cours d'introduction à JavaScript avant d'aborder ce contenu avancé.

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Debug VScode

Debug VScode

Le débogage est essentiel pour localiser et corriger des erreurs dans un programme. JavaScript, étant un langage très permissif, cache parfois des erreurs silencieuses. Savoir utiliser les outils adaptés permet d'économiser un temps précieux.

Visual Studio Code offre un support riche pour le débogage de différents types d'applications. Il intègre nativement des outils pour déboguer du JavaScript, du TypeScript et des applications Node.js

Rappels

Éléments de premier ordre 1/2

En JavaScript, on dit que les fonctions sont des éléments de premier ordre (ou citoyens de première classe – first-class citizens). Cela signifie qu'elles sont traitées comme n'importe quelle autre valeur dans le langage.

Éléments de premier ordre 2/2

- Les fonctions peuvent être stockées dans des variables
- Les fonctions peuvent être passées en argument à d'autres fonctions
- Les fonctions peuvent être retournées par d'autres fonctions
- Les fonctions peuvent être stockées dans des structures de données (objets, tableaux, etc.)

La portée des données (scope) 1/2

`var` est hoisté mais non bloqué

Les variables déclarées avec var sont hoistées (remontées) en haut de leur fonction ou du scope global, ce qui signifie qu'elles existent avant même leur ligne de déclaration, mais avec une valeur initiale undefined.

De plus, var n’a pas de portée de bloc, donc une variable déclarée dans un if, une boucle, ou tout autre bloc est accessible en dehors de ce bloc, ce qui peut entraîner des comportements inattendus.

La portée des données (scope) 2/2

`let` et `const` sont bloqués dans leur scope

Contrairement à var, les variables déclarées avec let ou const sont également hoistées, mais non initialisées. Elles entrent dans une zone dite "temporal dead zone" (zone morte temporaire) jusqu'à leur déclaration effective. Toute tentative d'accès avant la déclaration déclenche une erreur.

De plus, elles respectent la portée de bloc, donc ne sont accessibles que dans le bloc {} dans lequel elles sont définies.

La portée des fonctions en JavaScript 1/3

Fonctions déclarées (function declaration)

- Hoistées : elles sont déplacées en haut de leur scope (comme var) → tu peux les appeler avant leur définition.
- Portée fonctionnelle ou globale : une fonction déclarée en dehors de toute autre fonction est globale, sinon elle est locale à la fonction dans laquelle elle est déclarée.

La portée des fonctions en JavaScript 2/3

Fonctions anonymes affectées à une variable (function expression)

- Pas hoistées (ou plutôt, seule la variable est hoistée, pas sa valeur).
- Impossible de les appeler avant leur définition.
- Respectent la portée du bloc dans lequel elles sont définies (let ou const).

La portée des fonctions en JavaScript 3/3

Fonctions imbriquées

Une fonction définie à l'intérieur d'une autre a accès à toutes les variables de la fonction englobante, grâce à la portée lexicale.

Objets ou fonctions ?

Tout est objet sauf les types primitifs. Les fonctions peuvent contenir des propriétés :

```
function demo() {}
demo.prop = 123;
console.log(demo.prop); // 123
```

Pièges classiques du langage

Typage faible

```
console.log("5" + 1); // "51"
console.log("5" - 1); // 4
```

Le moteur convertit automatiquement les types, ce qui peut être source d'erreurs.

Pièges classiques du langage

Hoisting

```
console.log(message); // undefined
var message = "Bonjour";
```

Les variables `var` sont déclarées en haut du scope, mais pas initialisées.

Pièges classiques du langage

Variables globales implicites

```
function test() {
  x = 10; // devient globale si non déclarée avec let/const/var
}
```

Pièges classiques du langage

Changement de contexte

```
let obj = {
  valeur: 42,
  afficher: function() {
    console.log(this.valeur);
  }
};
let f = obj.afficher;
f(); // undefined car dans un autre contexte
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

POO

Programmation orientée objet en JavaScript

En JS, tout est objet. Plusieurs façons de créer des objets existent :

Objet littéral

```
let voiture = {
  marque: "Toyota",
  demarrer() {
    console.log("Démarrage...");
  }
};
```

Programmation orientée objet en JavaScript

Fonction constructeur

```
function Voiture(marque) {
  this.marque = marque;
  this.demarrer = function() {
    console.log("Démarrage de la " + this.marque);
  }
}
```

Programmation orientée objet en JavaScript

Object.create

```
let machine = {
  allumer() {
    console.log("Machine allumée");
  }
};
let robot = Object.create(machine);
robot.nom = "RX-1";
```

`this` et le contexte

```
let obj = {
  valeur: 42,
  normal: function() {
    console.log(this.valeur);
  },
  flechee: () => {
    console.log(this.valeur);
  }
};
obj.normal(); //42
obj.flechee(); // undefined
```

Explication :

- `function() {}` a un `this` dynamique
- `() => {}` garde le `this` du contexte parent

Prototype et `__proto__`

Exemple de prototype :

```
function Personne(nom) {
  this.nom = nom;
}
Personne.prototype.saluer = function() {
  console.log("Bonjour, je m'appelle " + this.nom);
};
let p = new Personne("Alice");
p.saluer();
```

Vérification du lien prototype :

```
console.log(p.__proto__ === Personne.prototype); // true
```

Héritage

Via prototype :

```
function Animal(nom) {
  this.nom = nom;
}
Animal.prototype.parler = function() {
  console.log(this.nom + " fait du bruit");
};

function Chien(nom) {
  Animal.call(this, nom);
}
Chien.prototype = Object.create(Animal.prototype);
Chien.prototype.constructor = Chien;
```

Héritage

Via ES6 classes :

```
class Animal {
  constructor(nom) {
    this.nom = nom;
  }
  parler() {
    console.log(this.nom + " fait du bruit");
  }
}
class Chien extends Animal {
  parler() {
    console.log(this.nom + " aboie");
  }
}
```

Visibilité

- Tout est public par défaut
- Depuis ES2022 : `#propriete` = privée

Exemple

```
class Compte {
  #solde = 0;
  deposer(montant) {
    this.#solde += montant;
  }
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Programmation fonctionnelle

Programmation fonctionnelle

La programmation fonctionnelle repose sur des fonctions pures, des appels sans effets de bord, et des concepts comme les closures.

Fonctions anonymes

Fonctions sans nom, utiles pour des appels temporaires :

```
setTimeout(function() {
  console.log("Ceci est une fonction anonyme");
}, 1000);
```

Programmation fonctionnelle

Fonctions immédiates (IIFE)

S’exécutent immédiatement après leur déclaration :

```
(function() {
  console.log("Exécutée immédiatement");
})();
```

Programmation fonctionnelle

Fonctions internes

Fonctions définies à l'intérieur d'autres fonctions. Utiles pour limiter la portée :

```
function externe() {
  function interne() {
    console.log("Interne");
  }
  interne();
}
```

Programmation fonctionnelle

Redéfinition

Une fonction peut être redéfinie à la volée :

```
function direBonjour() {
  console.log("Bonjour");
}
direBonjour = function() {
  console.log("Salut");
}
direBonjour();
```

Programmation fonctionnelle

`arguments` et pseudo-surcharge

Le mot-clé `arguments` permet d'accepter un nombre indéfini de paramètres :

```
function addition() {
  let total = 0;
  for(let i = 0; i < arguments.length; i++) {
    total += arguments[i];
  }
  return total;
}
```

Programmation fonctionnelle

Closure

Une closure permet de créer une fonction avec un état "privé" persistant :

```
function compteur() {
  let c = 0;
  return function() {
    c++;
    console.log(c);
  }
}
let monCompteur = compteur();
monCompteur(); // 1
monCompteur(); // 2
```

Explication : la fonction retournée garde accès à `c`, même après l'exécution de `compteur`.

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Programmation asynchrone

Programmation asynchrone

La programmation asynchrone permet d'exécuter des tâches **sans bloquer** le reste du programme.

En JavaScript, c'est essentiel car le langage fonctionne sur un **seul thread** : une seule opération peut s'exécuter à la fois.

Donc, au lieu d'attendre qu'une tâche longue (comme une requête réseau, un accès disque ou un minuteur) se termine, **le programme continue à s'exécuter**, et la tâche prévient quand elle est terminée.

Promesses et async/await

```
function attendre(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function demo() {
  console.log("Début");
  await attendre(1000);
  console.log("Fin après 1s");
}
demo();
```

Explication : `await` attend que la promesse se résolve, sans bloquer l'exécution globale.

Récupération avec `fetch`

```
async function chargerUtilisateurs() {
  let reponse = await fetch("https://jsonplaceholder.typicode.com/users");
  let donnees = await reponse.json();
  console.log(donnees);
}
```

Astuce : toujours gérer les erreurs avec `try/catch`.

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Structuration et qualité du code

Structuration et qualité du code

- Structurer proprement une application en plusieurs fichiers JavaScript
- Apprendre à utiliser les modules (ESM)
- Appréhender l'importance de la qualité de code avec les outils de linting

Modularisation

Un code bien organisé est plus facile à maintenir et à faire évoluer. La séparation en fichiers/modules est cruciale en développement professionnel.

Deux grands systèmes de modules existent en JavaScript :

- CommonJS (CJS) — basé sur require et module.exports
- ECMAScript Modules (ESM) — basé sur import et export

Modularisation

Aspect	CommonJS (`require`)	ESM (`import/export`)
Standard	Non (Node.js)	Oui (ECMAScript officiel)
Chargement	Synchrone	Asynchrone
Tree-shaking	❌ Non supporté	✅ Supporté
Compatibilité navigateur	❌ Non sans bundler	✅ Natif dans les navigateurs modernes
Utilisation	Simplicité (Node.js)	Modernité et portabilité
Syntaxe dynamique	✅ (peut faire `require(variable)`)	❌ (statique uniquement)

Modularisation

Démonstration avec `export` / `import`

```
// panier.js
export class Panier {
  constructor() {
    this.produits = [];
  }
  ajouter(nom, prix) {
    this.produits.push({ nom, prix });
  }
  total() {
    return this.produits.reduce((s, p) => s + p.prix, 0);
  }
}
```

Programme principal :

```
// main.js
import { Panier } from './panier.js';
const p = new Panier();
p.ajouter("Stylo", 1.5);
console.log(p.total());
```

3. Impact des closures sur la lisibilité

Trop de fonctions imbriquées avec des variables capturées peuvent rendre le code illisible :

```
function exterieur() {
  let secret = "valeur";
  return function() {
    return function() {
      console.log(secret);
    }
  }
}
```

Utiliser ces techniques avec parcimonie pour éviter de créer un "spaghetti de fonctions".

JSHint et JSLint : analyseurs statiques pour JavaScript

Outil	Description
JSLint	Créé par **Douglas Crockford** (aussi connu pour avoir popularisé JSON), JSLint est un **outil d'analyse statique de code JavaScript**. Il examine le code à la recherche d'erreurs, de mauvaises pratiques et de violations de style. **Très strict**, il impose des conventions rigides et **n'est pas configurable**. Il est destiné à promouvoir une écriture de code très propre, mais peut être perçu comme trop contraignant.
JSHint	JSHint est un **fork de JSLint**, créé pour répondre au besoin de **flexibilité**. Contrairement à JSLint, JSHint est **hautement configurable** : tu peux choisir les règles à appliquer ou ignorer. Il est donc plus adapté à des équipes ou des projets ayant des conventions de style propres. Il vérifie aussi la présence d'erreurs potentielles, tout en étant plus permissif.

Démonstration en ligne

Aller sur https://jshint.com, coller :

```
function test() {
  a = 5; // non déclaré !
  var b = 10;
  return a + b;
}
```

Exemple d'erreurs détectées :

- Variables non déclarées
- Syntaxes obsolètes
- Utilisation de `var` au lieu de `let` / `const`

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

API HTML5 en JavaScript

Validation des formulaires

La validation des données utilisateur est essentielle pour éviter les erreurs ou failles de sécurité. HTML5 permet des validations de base, mais JavaScript permet d'aller plus loin.

```
<form id="form">
  <input type="email" id="email" required />
  <button type="submit">Envoyer</button>
</form>
<script>
  document.getElementById("form").addEventListener("submit", function(e) {
    e.preventDefault();
    const email = document.getElementById("email").value;
    if (!email.includes("@")) {
      alert("Email invalide");
    } else {
      alert("Envoyé : " + email);
    }
  });
</script>
```

Stockage local (localStorage)

`localStorage` permet de stocker des paires clé/valeur sous forme de chaînes de caractères, de manière persistante dans le navigateur.

```
localStorage.setItem("prenom", "Alice");
console.log(localStorage.getItem("prenom")); // Alice
localStorage.removeItem("prenom");
```

Stockage structuré (IndexedDB)

Stocker de grandes quantités de données côté client (ex : objets complexes, bases locales, données hors ligne)

```
let request = indexedDB.open("MaBase", 1);
request.onupgradeneeded = function(e) {
  let db = e.target.result;
  db.createObjectStore("utilisateurs", { keyPath: "id" });
};
```

JSON et stockage

`localStorage` ne stocke que des chaînes. Pour enregistrer des objets :

```
const utilisateur = { nom: "Jean", age: 30 };
localStorage.setItem("utilisateur", JSON.stringify(utilisateur));
const recup = JSON.parse(localStorage.getItem("utilisateur"));
console.log(recup.nom); // Jean
```

WebSockets

WebSocket JavaScript définition :

Le WebSocket est un protocole de communication bidirectionnel et en temps réel entre un client (navigateur) et un serveur. Contrairement au protocole HTTP, qui suit un modèle requête-réponse, une connexion WebSocket reste ouverte, permettant au serveur d'envoyer des données au client à tout moment sans que celui-ci ait besoin de faire une nouvelle requête.

Exemple WebSockets

```
let socket = new WebSocket("ws://echo.websocket.org");
socket.onopen = () => socket.send("Hello Server");
socket.onmessage = (event) => console.log("Message :", event.data);
```

WebWorkers

Web Worker JavaScript définition:

Un Web Worker est une fonctionnalité de JavaScript qui permet d'exécuter du code dans un thread séparé (en arrière-plan) sans bloquer le fil principal de l'application (le main thread, où s'exécutent le DOM et l'interface utilisateur). C’est utile pour effectuer des tâches lourdes (comme des calculs complexes ou du traitement de données) sans figer l’interface.

Les Web Workers n'ont pas accès au DOM (document, window, alert, etc.), mais peuvent utiliser des APIs comme fetch, setTimeout, et XMLHttpRequest.

Exemple WebWorkers

```
// worker.js
onmessage = function(e) {
  postMessage(e.data * 2);
};
```

```
// main.js
let worker = new Worker("worker.js");
worker.postMessage(10);
worker.onmessage = function(e) {
  console.log("Résultat:", e.data); // 20
};
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

JavaScript et Node.js

JavaScript et Node.js

Node.js – Définition

Node.js est un environnement d'exécution JavaScript côté **serveur**, basé sur le moteur **V8** de Chrome (le même qui exécute JavaScript dans le navigateur). Il permet d'exécuter du JavaScript **en dehors du navigateur**, notamment pour créer des serveurs, des outils en ligne de commande, des API, ou même des applications de bureau (avec Electron).

En résumé : Node.js permet à JavaScript de sortir du navigateur pour devenir un langage **full-stack**, aussi bien côté client que côté serveur.

Différence entre Node.js et JavaScript dans le navigateur

Caractéristique	JavaScript (navigateur)	Node.js
Environnement	Navigateur web (Chrome, Firefox...)	Serveur (via Node.js installé)
Accès au DOM	✅ Oui (document, window, etc.)	❌ Non
Accès au fichiers	❌ Non	✅ Oui (`fs`, `path`, etc.)
Utilisation principale	Manipulation d'interface utilisateur	Création de serveurs, scripts backend
Modules	`import/export` (ESM), ou global	`require` / `import` (CommonJS ou ESM)
APIs spécifiques	`fetch`, `localStorage`, `alert`, etc.	`http`, `fs`, `process`, etc.
Asynchrone	✅ (promesses, async/await)	✅ (même modèle, avec plus de possibilités)

Asynchronisme avec Node.js

1. Les callbacks (anciens mais encore utilisés)

```
const fs = require('fs');

fs.readFile('fichier.txt', 'utf8', (err, data) => {
  if (err) return console.error(err);
  console.log(data);
});
```

Asynchronisme avec Node.js

2. Les promesses (Promises)

```
const fs = require('fs').promises;

fs.readFile('fichier.txt', 'utf8')
  .then(data => {
    console.log(data);
  })
  .catch(err => {
    console.error(err);
  });
```

Asynchronisme avec Node.js

3. `async/await` (au-dessus des Promises)

```
const fs = require('fs').promises;

async function lire() {
  try {
    const data = await fs.readFile('fichier.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}

lire();
```

Asynchronisme avec Node.js

4. Les événements (EventEmitter / Streams)

Exemple avec un `setTimeout` asynchrone basé sur un événement :

```
const EventEmitter = require('events');

const em = new EventEmitter();

em.on('fini', (message) => {
  console.log('Événement reçu :', message);
});

setTimeout(() => {
  em.emit('fini', 'Traitement terminé après 2 secondes');
}, 2000);
```

Styles d'asynchronicité :

Méthode	Avantages	Inconvénients
Callback	Simple, natif	Illisible si imbriqué (callback hell)
Promesse	Chaine claire	Un peu verbeux sans `async/await`
async/await	Super lisible, moderne	Besoin de `try/catch`
EventEmitter	Parfait pour flux d'événements	Moins direct que les autres

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Evolution ES7 à ES14

ES7 (2016)

1. **`Array.prototype.includes()`**
   Vérifie si un tableau contient une valeur.

```
[1, 2, 3].includes(2); // true
```

2. **Opérateur d'exponentiation (`**`)**

```
2 ** 3; // 8
```

ES8 (2017)

1. `Object.values()` / `Object.entries()`

```
const obj = { a: 1, b: 2 };
Object.values(obj); // [1, 2]
Object.entries(obj); // [['a', 1], ['b', 2]]
```

2. `String.prototype.padStart()` / `padEnd()`

```
'5'.padStart(3, '0'); // '005'
```

3. `async` / `await`

   Syntaxe moderne pour l'asynchronisme.

```
async function run() {
  const res = await fetch('/data');
  const json = await res.json();
  console.log(json);
}
```

ES9 (2018)

1. **Rest/spread sur objets**

```
const { a, ...rest } = { a: 1, b: 2, c: 3 };
console.log(rest); // { b: 2, c: 3 }
```

2. **Promise.prototype.finally()**

```
fetch('/data')
  .then(res => res.json())
  .catch(err => console.error(err))
  .finally(() => console.log('Terminé'));
```

3. **RegEx améliorations :** `dotAll` (`s`), `named capture groups`, etc.

ES10 (2019)

1. `Array.prototype.flat()` / `flatMap()`

```
[1, [2, [3]]].flat(2); // [1, 2, 3]
```

2. `Object.fromEntries()`

```
Object.fromEntries([['a', 1], ['b', 2]]); // { a: 1, b: 2 }
```

3. `trimStart()` / `trimEnd()`

```
'   hello'.trimStart(); // 'hello'
```

ES11 (2020)

1. **Nullish coalescing (`??`)**

```
null ?? 'default'; // 'default'
```

2. **Optional chaining (`?.`)**

```
const user = {};
console.log(user?.profile?.email); // undefined sans erreur
```

3. **Promise.allSettled()**

```
Promise.allSettled([Promise.resolve(1), Promise.reject('erreur')])
```

4. **`globalThis`** — référence globale universelle (Node, browser, etc.)

ES12 (2021)

1. ## Logical assignment operators

```
let a = null;
a ??= 'valeur par défaut'; // 'valeur par défaut'
```

2. ## Numeric separators

```
const budget = 1_000_000; // plus lisible
```

3. ## `String.prototype.replaceAll()`

```
'foo-bar-bar'.replaceAll('bar', 'baz'); // 'foo-baz-baz'
```

4. ## `Promise.any()`

```
Promise.any([Promise.reject(), Promise.resolve('ok')]); // Résout avec 'ok'
```

ES13 (2022)

1. `at()` sur Array / String

```
[1, 2, 3].at(-1); // 3
'abc'.at(-1); // 'c'
```

2. Top-level `await` (dans les modules ES)

```
const data = await fetch('/api');
```

3. Class fields / méthodes privées

```
class Person {
  #secret = 'privé';
  getSecret() {
    return this.#secret;
  }
}
```

ES14 (2023)

1. `Array.prototype.toSorted()`, `toReversed()`, `toSpliced()` (non destructif)

```
const arr = [3, 1, 2];
const sorted = arr.toSorted(); // [1, 2, 3], sans modifier `arr`
```

2. `Symbol.prototype.description`

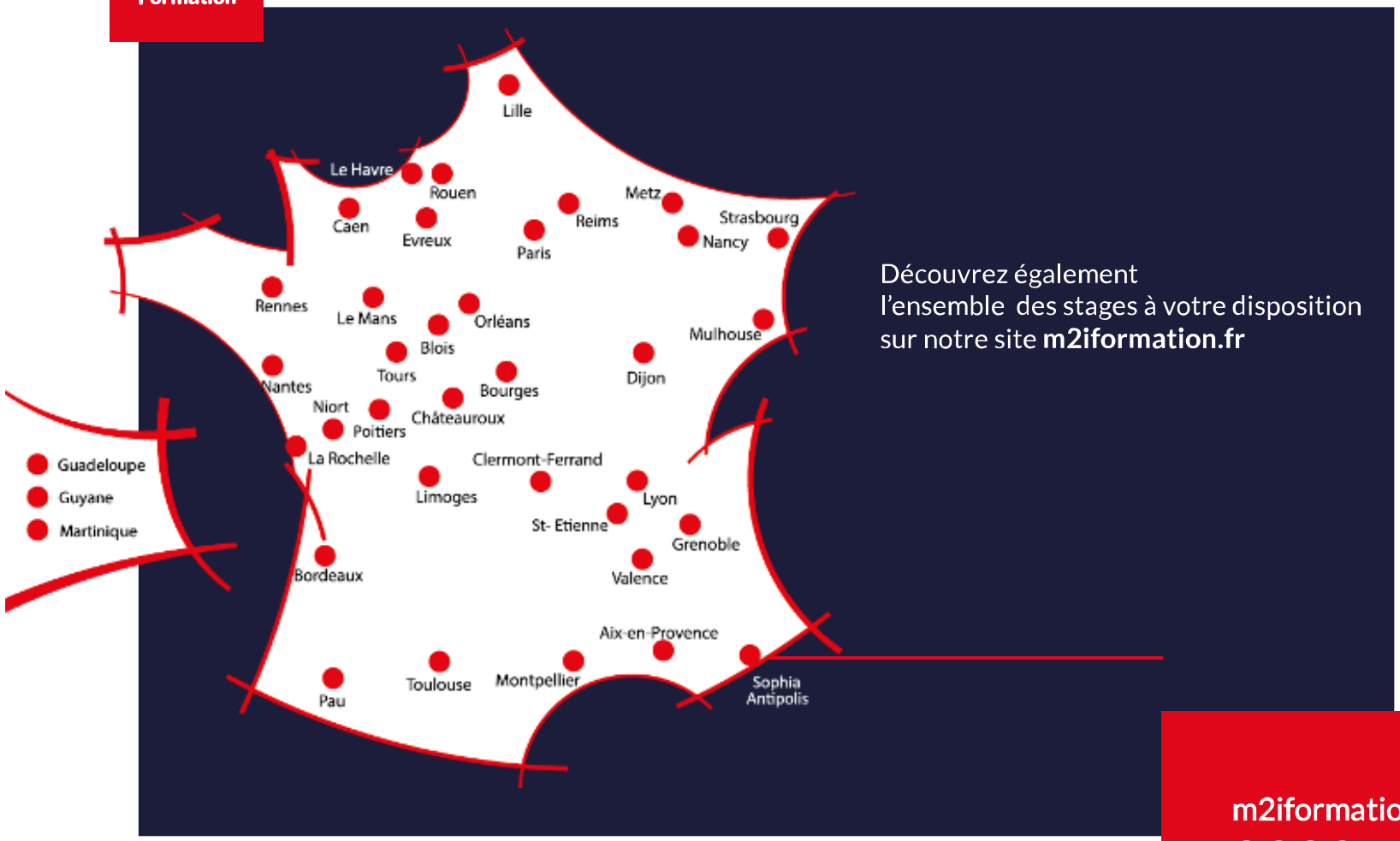
```
const sym = Symbol('hello');
console.log(sym.description); // 'hello'
```

3. **Améliorations** `Hashbang`

```
#!/usr/bin/env node
console.log('Script Node.js avec hashbang');
```

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr