

# Rappels ES6

- Structuration
- var et let
- Fonctions fléchées
- Les promesses

# ES ou ECMAScript

---

# Introduction à **ES6**

- Publié en 2015, avec de nombreuses évolutions par rapport à sa version précédente ES5
- Un standard pour un langage de programmation



**{ ES6 JS }**

- **Lien utile** : <https://htmlcheatsheet.com/js/>
- **Pour vous exercer** : <https://www.programiz.com/javascript/online-compiler/>

# Concepts fondamentaux

---

# Rappel : Opérateur **ternaire**

- Forme abrégée de l'instruction if...else

Uploaded using RayThis Extension

```
condition ? expressionSiVrai : expressionSiFaux;
```

Exemple ternaire

```
let nombre = 10;  
let resultat = (nombre % 2 === 0) ? "pair" : "impair";  
console.log(resultat); // Affiche "pair"
```

# ES6 : **let**, **var** et **const**

- Introduction des mots clés **let** et **const**
- **let** et **const** ajoutent une notion de portée

```

let, var et c'vons

function varTest() {
  var x = 1;
  if (true) {
    var x = 2; // c'est la même variable !
    console.log(x); // 2
  }
  console.log(x); // 2
}

function letTest() {
  let x = 1;
  if (true) {
    let x = 2; // c'est une variable différente
    console.log(x); // 2
  }
  console.log(x); // 1
}

function constTest() {
  const x = 1;
  if (true) {
    const x = 2; // c'est une variable différente
    console.log(x); // 2
  }
  console.log(x); // 1

  // x = 3; // Erreur : Assignment to constant variable.
}

varTest();
letTest();
constTest();
  
```

# ES6 : manipulation des **array**

- Une syntaxe plus concise qui simplifie le code (notamment dans les callbacks)

```
structuration

//array structuration
let arrayStructuration = [1, 2]
console.log(arrayStructuration); //[1, 2]

//array copy
let arrayCopy = [...arrayStructuration, 3, 4]
console.log(arrayCopy); //[1, 2, 3, 4]

//array destructuration
let [a, b] = arrayCopy;
console.log(a); //1
console.log(b); //2
```

```
structuration

//object structuration
let obj = {
  message: "hello world"
}
console.log(obj); //{message: "hello world"}

//object add key
obj.type = "success";
console.log(obj); //{message: "hello world", type: "success"}

//object copy
let copy = {...obj}

//object destructuration
const {message} = obj;
console.log(message); //"hello world"
```

# ES6 : les classes

- Avant sous forme de prototype
- Offre maintenant la possibilité d'utiliser des classes

```
Classes

// ES5
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.greet = function() {
  return "Hello, my name is " + this.name;
};

// ES6
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    return `Hello, my name is ${this.name}`;
  }
}
```



# ES6 : les fonctions fléchées

- Une syntaxe plus concise qui simplifie le code

```
Fonctions fléchées

// ES5
function foo() {
  console.log("bar");
}

// ES6
const foo = () => {
  console.log("bar");
}

// ou encore plus concis ES6
const foo = () => console.log("bar");
```

```
Fonctions fléchées

// ES5
const array = [1, 2, 3];

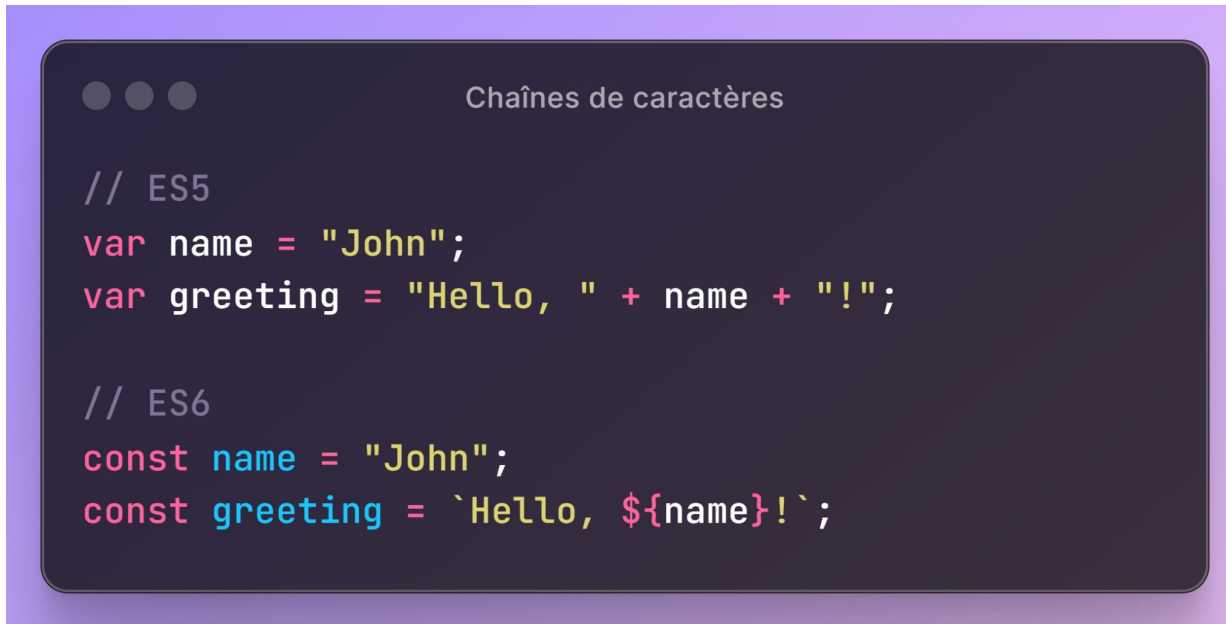
for (let i = 0; i < array.length; i++) {
  console.log(array[i]);
}

// ES6
const array = [1, 2, 3];

array.map(i => console.log(i));
```

# ES6 : les chaînes de caractères

- Une syntaxe plus concise qui simplifie le code



```

// ES5
var name = "John";
var greeting = "Hello, " + name + "!";

// ES6
const name = "John";
const greeting = `Hello, ${name}!`;

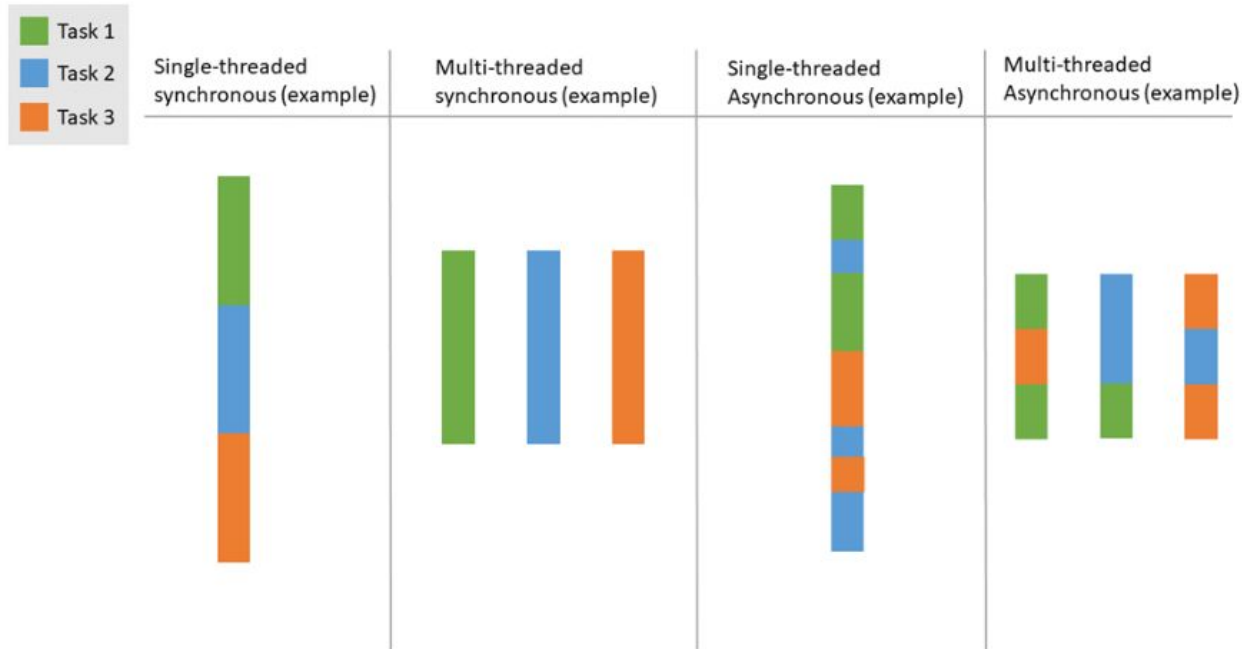
```

# Les Threads

---

# Notion de **thread**

- Unité d'exécution dans un processus qui permet d'effectuer plusieurs tâches en parallèle
- Permet le multitâche, la réactivité, mais souvent plus complexe et dépendant du reste



# Les promesses

---

# ES6 : les **promesses**

- Fonctionnalité introduite dans ES6 pour gérer les opérations asynchrones de manière plus propre et plus lisible qu'avec les callbacks traditionnels
- Les callbacks sont des fonctions passées en argument à d'autres fonctions et exécutées après que l'opération principale est terminée.
- Une fonction avec `async` retourne une promesse.
- `await` est utilisé pour attendre la résolution de cette promesse

```

Promesses

// ES5, utilisation de callback
function sleep(ms, callback) {
  setTimeout(callback, ms);
}

function asynchroneFunction1() {
  console.log("startEs5");
  sleep(3000, function() {
    console.log("endEs5"); // exécuté après 3 secondes
  });
}

asynchroneFunction1();

// ES6 : promesse
const sleep = (ms) => new Promise(resolve => setTimeout(resolve, ms));

const asynchroneFunction2 = () => {
  console.log("startEs6");
  sleep(3000).then(() => {
    console.log("endEs6"); // exécuté après 3 secondes
  });
};

asynchroneFunction2();

// ES8 : await / async
const sleep = (ms) => new Promise(resolve => setTimeout(resolve, ms));

const asynchroneFunction3 = async () => {
  console.log("startEs8");
  await sleep(3000);
  console.log("endEs8"); // exécuté après 3 secondes
};

asynchroneFunction3();

```

# ES6 : les **promesses**

- Après 3 secondes ou immédiatement ?

```
Uploaded using RayThis Extension

const sleep = (ms) => new Promise(resolve => setTimeout(resolve, ms));

const asynchroneFunction = async () => {
  console.log("start");
  await sleep(3000);
  console.log("end");
};

asynchroneFunction();
```

```
Uploaded using RayThis Extension

const sleep = (ms) => new Promise(resolve => setTimeout(resolve, ms));

const asynchroneFunction2 = () => {
  console.log("startEs6");
  sleep(3000)
  console.log("endEs6");
};

asynchroneFunction2();
```

# ES6 : promesses , gestion d'erreur

- resolve et reject utilisés pour marquer une promesse comme résolue ou la rejeter
- try/catch utilisé pour la gestion d'erreurs

```

// ES6
asyncFunction()
  .then(result => {
    console.log('Résultat :', result);
  })
  .catch(error => {
    console.error('Erreur :', error.message);
  });
  
```

```

// ES6
function asyncFunction() {
  return new Promise((resolve, reject) => {
    // Simulation d'une opération asynchrone
    setTimeout(() => {
      const randomNumber = Math.random();

      if (randomNumber < 0.5) {
        // Résoudre la promesse avec le nombre aléatoire
        resolve(randomNumber);
      } else {
        // Rejeter la promesse avec une erreur
        reject(new Error('Une erreur s\'est produite !'));
      }
    }, 1000);
  });
}

// ES8
async function executeAsyncFunction() {
  try {
    const result = await asyncFunction();
    console.log('Résultat :', result);
  } catch (error) {
    console.error('Erreur :', error.message);
  }
}
  
```



# ES6 : promesses, exemple “waterfall”

- Exécution séquentielle d'opérations asynchrones.
- Chaque opération dépend du résultat de la précédente

```
Uploaded using RayThis Extension

async function asyncOperation1() { /* ... */ }
async function asyncOperation2() { /* ... */ }
async function asyncOperation3() { /* ... */ }

async function executeAsyncOperations() {
  try {
    const result1 = await asyncOperation1();
    const result2 = await asyncOperation2(result1);
    const result3 = await asyncOperation3(result2);
    console.log('Résultat final :', result3);
  } catch (error) {
    console.error('Erreur :', error);
  }
}

executeAsyncOperations();
```

# ES6 : promesses, exemple “parallèle”

- Exécution parallèle d'opérations asynchrones.
- Toutes les opérations sont lancées en même temps, et les résultats sont traités ensemble.

```

    Uploaded using RayThis Extension

    async function asyncOperation1() { /* ... */ }
    async function asyncOperation2() { /* ... */ }
    async function asyncOperation3() { /* ... */ }

    async function executeParallelOperations() {
      try {
        const [result1, result2, result3] = await Promise.all([
          asyncOperation1(),
          asyncOperation2(),
          asyncOperation3()
        ]);
        console.log('Résultat 1 :', result1);
        console.log('Résultat 2 :', result2);
        console.log('Résultat 3 :', result3);
      } catch (error) {
        console.error('Erreur :', error);
      }
    }

    executeParallelOperations();
  
```



# Exercices - ES6

---



m2information.fr