Les fonctions en JavaScript

> En JavaScript, les fonctions sont des **blocs de code réutilisables** conçus pour **exécuter une tâche spécifique**.
> Elles jouent un **rôle central** dans l'**organisation**, la **modularité** et la **maintenance** des programmes.

1. Qu'est-ce qu'une fonction ?

> Une fonction est un **ensemble d'instructions** regroupées sous un même nom afin d'être **réutilisées** autant de fois que nécessaire.
> Elle peut recevoir des **paramètres** en entrée, effectuer une opération, et retourner éventuellement un **résultat**.

Exemple basique :

```javascript
function saluer() {
  console.log("Bonjour !");
}
saluer(); // Affiche "Bonjour !"
```

2. Syntaxe d'une fonction

2.1. Déclaration d'une fonction

> La **déclaration** de fonction utilise le mot-clé `function`, un nom de fonction, une liste de paramètres (facultative) et un corps d'exécution.

```javascript
function nomDeLaFonction(param1, param2) {
  // Corps de la fonction
  return valeur; // (facultatif)
}
```

Exemple :

```javascript
function addition(a, b) {
  return a + b;
}

console.log(addition(3, 4)); // 7
```

2.2. Appel d'une fonction

> Pour exécuter une fonction, il suffit de l'**appeler** avec son nom suivi de parenthèses contenant éventuellement des arguments.

```javascript
nomDeLaFonction(arg1, arg2);
```

3. Types de fonctions en JavaScript

3.1. Fonctions déclarées

> Les **fonctions déclarées** sont définies avec le mot-clé `function` et peuvent être utilisées **avant ou après** leur définition (`hoisting`).

Exemple : déclaration de fonction **avant** l'appel

```javascript
function multiplier(a, b) {
  return a * b;
}
console.log(multiplier(2, 3)); // 6
```

Exemple : déclaration de fonction **après** l'appel

```javascript
console.log(diviser(10, 2)); // 5

function diviser(a, b) {
  return a / b;
}
```

3.2. Fonctions expressions

> Les **fonctions expressions** sont affectées à une variable. .
> Elles ne sont accessibles **qu'après leur déclaration**.

Exemple : fonction expression **anonyme**

```javascript
const diviser = function(a, b) {
  return a / b;
};
console.log(diviser(10, 2)); // 5
```

3.3. Fonctions fléchées (arrow functions)

> Introduites avec ES6, elles offrent une syntaxe plus concise.
> Elles sont souvent utilisées pour des fonctions anonymes ou comme callback.
>
> Elles ne possèdent pas leur propre `this`, ce qui les rend idéales pour des méthodes de classe ou des fonctions imbriquées.

Exemple : aucun paramètre

```javascript
const direBonjour = () => console.log("Bonjour !");
direBonjour(); // "Bonjour !"
```

Exemple : un seul paramètre

```javascript
const doubler = x => x * 2;
console.log(doubler(5)); // 10
```

Exemple : plusieurs paramètres

```javascript
const soustraire = (a, b) => a - b;
console.log(soustraire(9, 4)); // 5
```

3.4. Fonctions anonymes

> Une fonction anonyme n'a pas de nom.
>
> Elle est souvent utilisée comme fonction de rappel (`callback`).

Exemple : fonction anonyme en tant que callback dans `setTimeout`

```javascript
setTimeout(function() {
  console.log("Temps écoulé !");
}, 1000);
```

3.5. Fonctions immédiatement invoquées (IIFE)

> Une IIFE est une fonction qui s'exécute directement après sa déclaration.

Exemple :

```javascript
(function() {
  console.log("Cette fonction s'exécute immédiatement !");
})();
```

4. Paramètres et arguments

4.1. Paramètres et leur utilisation

> Les **paramètres** sont des variables définies **dans la déclaration** de la fonction.
> Les **arguments** sont les valeurs passées à la fonction **lors de l'appel**.

Exemple :

```javascript
function direBonjour(nom) {
  console.log(`Bonjour, ${nom} !`);
}

direBonjour("Alice"); // "Bonjour, Alice !"
```

4.2. Paramètres par défaut

> Les **paramètres par défaut** permettent de définir une **valeur initiale** lorsqu'**aucun argument** n'est passé.

Exemple :

```javascript
function salutation(nom = "invité") {
  console.log(`Bienvenue, ${nom} !`);
}

salutation(); // "Bienvenue, invité !"
salutation("Bob"); // "Bienvenue, Bob !"
```

4.3. Paramètres nommés (objets)

> Les **paramètres nommés** sont souvent utilisés pour passer des objets à une fonction, permettant ainsi de **spécifier uniquement les propriétés nécessaires**.

Exemple :

```javascript
function afficherPersonne({ nom, age }) {
  console.log(`Nom : ${nom}, Âge : ${age}`);
}

afficherPersonne({ nom: "Alice", age: 30 }); // "Nom : Alice, Âge : 30"
```

4.4. L'objet arguments

L'objet `arguments` est une propriété de toutes les fonctions, qui contient tous les arguments passés à la fonction, même si elle n'a pas de paramètres définis.
Il est non standard et ne doit pas être utilisé dans les fonctions fléchées.
Il est préférable d'utiliser les paramètres rest pour obtenir un tableau d'arguments.

Exemple :

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

console.log(somme(1, 2, 3, 4)); // 10
```

4.5. Rest parameters

Les paramètres rest permettent de regrouper plusieurs arguments dans un tableau.

Exemple :

```javascript
function somme(...nombres) {
  return nombres.reduce((total, nombre) => total + nombre, 0);
}
console.log(somme(1, 2, 3, 4)); // 10
```

5. Valeurs retournées

Une fonction peut retourner une valeur avec le mot-clé `return`.

Si aucune valeur n'est retournée, la fonction retourne `undefined` par défaut.

Si une fonction ne contient pas de `return`, elle exécute son code et se termine sans retourner de valeur.

Il est possible d'utiliser `return` sans valeur pour sortir d'une fonction prématurément.

Tout code écrit après un `return` dans une fonction ne sera pas exécuté.

Exemple : valeur de retour standard

```javascript
function cube(x) {
  return x ** 3;
}

console.log(cube(3)); // 27
```

Exemple : absence du mot clé `return`

```javascript
function carre(x) {
  x ** 2;
}

console.log(carre(3)); // undefined
```

Exemple : sortie prématurée

```javascript
function verifierPair(x) {
  if (x % 2 == 0) {
    return; // Sortie prématurée
  }
  console.log(`${x} est pair`);
}

verifierPair(4); // "4 est pair"
verifierPair(5); // undefined
```

Exemple : code après le `return` non exécuté

```javascript
function exemple() {
  return "Bonjour";
  console.log("Ceci ne sera pas affiché");
}
console.log(exemple()); // "Bonjour"
```

6. Scope et fermetures (`closures`)

6.1. Scope (portée)

> Le scope d'une variable désigne la zone où elle est accessible.

Il existe deux types principaux de scope :

- Local : Déclaré à l'intérieur d'une fonction.
- Global : Déclaré en dehors de toute fonction.

Exemple :

```javascript
const variableGlobale = "Je suis global";

function exempleScope() {
  const variableLocale = "Je suis local";
  console.log(variableGlobale); // Accessible ici
}
// console.log(variableLocale); // Erreur : Non définie
```

6.2. Closures

> Une `closure` se produit lorsque les variables d'une fonction sont conservées même après que la fonction ait terminé son exécution.

Exemple :

```javascript
function creerCompteur() {
  let compteur = 0;
  return function() {
    compteur++;
    return compteur;
  };
}

const incrementer = creerCompteur();
console.log(incrementer()); // 1
console.log(incrementer()); // 2
```

7. Fonctions d'ordre supérieur (`higher-order functions`)

> Les fonctions peuvent accepter d'autres fonctions comme arguments ou en retourner.
> On les appelle fonctions d'ordre supérieur.

Exemple avec un callback :

```javascript
function executerDeuxFois(fonction) {
  fonction();
  fonction();
}

executerDeuxFois(() => console.log("Hello"));
// Affiche "Hello" deux fois
```

Exemple d'une fonction qui retourne une fonction :

```javascript
function createMultiplier(facteur) {
  return function(nombre) {
    return nombre * facteur;
  };
}

const doubler = createMultiplier(2);
console.log(doubler(5)); // 10
```

8. Cas d'utilisation courants

8.1. Manipulation de tableaux avec `map`, `filter`, `reduce`

```javascript
const nombres = [1, 2, 3, 4];

const doubles = nombres.map(x => x * 2);
console.log(doubles); // [2, 4, 6, 8]

const pairs = nombres.filter(x => x % 2 === 0);
console.log(pairs); // [2, 4]

const somme = nombres.reduce((acc, curr) => acc + curr, 0);
console.log(somme); // 10
```

8.2. Gestion des événements

```javascript
document.querySelector("button").addEventListener("click", () => {
  console.log("Bouton cliqué !");
});
```

9. À RETENIR

- Les fonctions déclarées peuvent être utilisées avant leur définition.
- Les fonctions expressions et fléchées s'appuient sur des variables et n'acceptent pas le `hoisting`.
- Les paramètres par défaut et les rest parameters offrent une grande flexibilité.
- Les scopes garantissent la sécurité des données locales d'une fonction.
- Les closures permettent de créer des fonctions avec état.
- Les fonctions d'ordre supérieur apportent une modularité et une lisibilité exceptionnelles.

La maîtrise des fonctions est un atout fondamental pour écrire un code clair et réutilisable en JavaScript.
Familiarisez-vous avec leurs divers types et possibilités pour enrichir vos projets avec des solutions puissantes et élégantes !