Les Chaînes de caractères en JavaScript

Introduction aux chaînes

Définition

Les **chaînes (string)** sont des ensembles de caractères qui représentent un **texte**.

Les chaînes de caractères sont des objets, et comme tout objet, elles ont des propriétés et des méthodes.

Création de chaînes

Simple guillemets (`''`)

```
let chaine1 = 'Bonjour !';
```

Les guillemets simples sont utilisés pour déclarer des chaînes de caractères en JavaScript.

Doubles guillemets (`""`)

```
let chaine2 = "Bonjour !";
```

Les doubles guillemets sont aussi utilisés pour créer une chaîne de caractères en JavaScript.

Backticks (`)

```
let chaine3 = `Bonjour !`;
```

Utilisation des Backticks:

- Permet de créer des **chaînes de caractères** avec des variables intégrées.
- Syntaxe : `` `texte ${variable} texte` ``.

Exemple :

```
let nom = "World";
let chaine1 = `Hello, ${nom}!`;
```

L'avantage des backticks est de pouvoir insérer des variables dans des chaînes de caractères, en évitant les opérations de concaténation.

Caractères spéciaux et échappement

Caractère	Description	Exemple
`\\\\`	Barre oblique inversée	`let a = "C:\\\\path";`
`\\'`	Apostrophe (guillemet)	`let b = 'It\\'s ok';`
`\\"`	Guillemet double	`let c = "Il a dit \\"Bonjour\\"";`
`\\n`	Saut de ligne	`let d = "Ligne 1\\nLigne 2";`
`\\t`	Tabulation	`let e = "Colonne 1\\tColonne 2";`

Exemple d'échappement :

```
let chaine = "Ceci est une \\"citation\\"";
```

L'échappement de caractères permet d'afficher des caractères spéciaux ou réservés dans une chaîne de caractères. Il est important de les utiliser lorsque c'est nécessaire pour éviter des erreurs ou un comportement inattendu de votre code.

Propriétés et méthodes de base

`length`

La propriété `length` représente la longueur d'une chaîne de caractères.

```
let chaine = "Bonjour le monde !";
let longueurChaine = chaine.length;

console.log(longueurChaine); // Résultat : 18
```

La propriété `length` peut également être utilisée pour déterminer la longueur d'autres types d'objets, tels que les tableaux.

Utilisation et exemples

```
const str = "Hello, World!";
const len = str.length; // 13
```

La propriété **length** est utilisée pour obtenir la longueur d'une chaîne de caractères.

`indexOf`

La méthode `indexOf()` renvoie la première occurrence d'une **sous-chaîne** dans une **chaîne**.

```
const phrase = "Bienvenue dans le cours de JavaScript !";

const indexMot = phrase.indexOf("cours");

console.log(indexMot); // Affiche 21, la position de "cours" dans la chaîne
```

Cette méthode est sensible à la casse, donc "COURS" et "cours" renverraient des indices différents.

Syntaxe

```
str.indexOf(searchValue, fromIndex)
```

`indexOf()` est une méthode utilisée pour chercher la première occurrence de `searchValue` dans la chaîne `str`, commençant à l'indice `fromIndex`. Si la valeur n'est pas trouvée, la méthode retourne -1.

Exemples d'utilisation

```
const str = "Hello, World!";
const index = str.indexOf("World"); // 7
const notFound = str.indexOf("Earth"); // -1
```

La méthode indexOf est utilisée pour trouver la première occurrence d'une chaîne de caractères spécifique dans une autre chaîne. Elle retourne l'indice de début de la chaîne recherchée ou -1 si la chaîne n'est pas trouvée.

`lastIndexOf`

La méthode `lastIndexOf()` renvoie la **dernière occurrence** d'une sous-chaîne dans une chaîne.

```
let chaine = "Javascript est un langage de programmation.";
let position = chaine.lastIndexOf("a");

console.log(position); // Affiche 29
```

Méthode	Description
`lastIndexOf()`	Trouve la dernière occurrence d'une sous-chaîne spécifiée dans une chaîne, et retourne sa position

Syntaxe

```
str.lastIndexOf(searchValue, fromIndex)
```

- **str** : la chaîne de caractères dans laquelle effectuer la recherche
- **searchValue** : la valeur à rechercher dans la chaîne de caractères
- **fromIndex** : l'indice à partir duquel commencer la recherche (facultatif, par défaut à la fin de la chaîne)

lastIndexOf() retourne l'indice de la dernière occurrence de searchValue dans str ou -1 si elle n'est pas trouvée.

Exemples d'utilisation

```
const str = "Hello, World! Goodbye, World!";
const index = str.lastIndexOf("World"); // 23
const notFound = str.lastIndexOf("Earth"); // -1
```

Modification de chaînes

`slice`

`slice` permet d'extraire une **sous-chaîne** d'une chaîne existante à partir d'indices de **début** et de **fin**.

```
let phrase = "Apprenez à utiliser la méthode slice.";

let sousChaine = phrase.slice(9, 24);

console.log(sousChaine); // "utiliser la méthode"
```

Paramètre	Description
début	Index de début où la sous-chaîne doit être extraite
fin	Index de fin, non inclus, jusqu'où extraire

- Si l'indice de fin n'est pas fourni, la méthode extrait jusqu'à la fin de la chaîne d'origine.

Il est important de rappeler que les indices commencent à 0.

Syntaxe

```
str.slice(beginIndex[, endIndex])
```

La méthode `.slice()` est utilisée pour extraire une partie d'une chaîne de caractères. Les paramètres `beginIndex` et `endIndex` sont optionnels et définissent les positions de début et de fin de l'extraction.

Exemples d'utilisation

```
let texte = "JavaScript";
let sousChaine = texte.slice(0, 4); // "Java"
```

La méthode slice permet d'extraire une partie d'une chaîne de caractères en indiquant l'indice de départ et l'indice de fin.

`substring`

`substring` est similaire à `slice`, mais avec quelques différences dans le comportement des **indices**.

```
const chaine = "JavaScript est incroyable !";

const resultat = chaine.substring(0, 10);

console.log(resultat); // Affiche "JavaScript"
```

Fonction	Indices négatifs	Inversion des indices
`slice`	Accepte	Non
`substring`	Non	Oui

Syntaxe

```
str.substring(indexStart[, indexEnd])
```

La méthode `substring` extrait les caractères d'une chaîne de caractères entre deux indices donnés, sans inclure le caractère à la position `indexEnd`. Si `indexEnd` n'est pas spécifié, elle prend par défaut la longueur totale de la chaîne.

Exemples d'utilisation

```
let texte = "JavaScript";
let sousChaine = texte.substring(0, 4); // "Java"
```

La fonction `substring()` est utilisée pour extraire une partie d'une chaîne de caractères en précisant les indices de début et de fin.

`split`

`split` divise une **chaîne** en un **tableau** de sous-chaînes basées sur un **séparateur**.

```
const phrase = 'Je suis un développeur JavaScript';
const mots = phrase.split(' ');

console.log(mots);
// Output: ["Je", "suis", "un", "développeur", "JavaScript"]
```

Le séparateur peut être aussi bien un caractère unique qu'une chaîne de caractères. Si aucun séparateur n'est fourni, la chaîne sera divisée par caractères.

Syntaxe

```
str.split([separator[, limit]])
```

La méthode `split` permet de diviser une chaîne de caractères à partir d'un séparateur et renvoie un tableau de sous-chaînes. Le paramètre optionnel `limit` permet de limiter le nombre de sous-chaînes obtenues.

Exemples d'utilisation

```
let texte = "JavaScript est super";
let mots = texte.split(" "); // ["JavaScript", "est", "super"]
```

`split()` est une méthode pour les chaînes de caractères qui découpe la chaîne en fonction du séparateur passé en argument et retourne un tableau des éléments découpés.

- Dans cet exemple, le séparateur est l'espace `" "`

`replace`

`replace` remplace une **sous-chaîne** ou un **motif** dans une chaîne par une nouvelle sous-chaîne ou la valeur retournée par une **fonction**.

Syntaxe	Description
`string.replace(regex, str2)`	Remplace par `str2` la première occurrence de `regex`
`string.replace(regex, func)`	Remplace par la valeur retournée par `func`
`string.replace(str1, str2)`	Remplace `str1` par `str2`
`string.replace(str1, func)`	Remplace `str1` par la valeur retournée par `func`

Préciser que lors de la présentation, il faut rappeler aux apprenants que `replace` ne modifie pas la chaîne d'origine, mais qu'elle renvoie une nouvelle chaîne modifiée.

Syntaxe

```
str.replace(regexp|substr, newSubstr|function)
```

La méthode replace() permet de remplacer une sous-chaîne de caractères ou une expression régulière par une nouvelle chaîne ou un résultat renvoyé par une fonction appelée.

Exemples d'utilisation

```
let texte = "JavaScript est super";
let nouveauTexte = texte.replace("super", "génial"); // "JavaScript est génial"
```

Cet exemple illustre comment utiliser la méthode **replace** pour remplacer un mot dans une chaîne de caractères.

Mise en forme des chaînes

`toLowerCase()`

La méthode `toLowerCase()` prend une **chaîne de caractères** et la **convertit en minuscules**.

```
const texte = "Convertir en minuscules";
const texteEnMinuscules = texte.toLowerCase();

console.log(texteEnMinuscules); // Résultat: "convertir en minuscules"
```

N'oubliez pas que `toLowerCase()` ne modifie pas la chaîne originale, elle crée une nouvelle chaîne avec les caractères en minuscules.

Syntaxe

```
let lowercaseString = originalString.toLowerCase();
```

La méthode `toLowerCase()` convertit une chaîne en minuscules.

Exemples d'utilisation

```
let message = "Hello World!";
let lower = message.toLowerCase(); // "hello world!"
```

La méthode toLowerCase() permet de convertir une chaîne de caractères en minuscules. Cette méthode est pratique pour standardiser les données texte avant leur traitement ou leur comparaison.

`toUpperCase()`

La méthode `toUpperCase()` prend une **chaîne de caractères** et la convertit en **majuscules**.

```
const originalString = "La programmation en JavaScript est amusante!";
const upperCaseString = originalString.toUpperCase();

console.log(upperCaseString); // Output: LA PROGRAMMATION EN JAVASCRIPT EST AMUSANTE!
```

`toUpperCase()` ne modifie pas la chaîne d'origine, elle retourne une nouvelle chaîne avec les caractères en majuscules.

Syntaxe

```
let uppercaseString = originalString.toUpperCase();
```

La méthode toUpperCase() permet de convertir une chaîne de caractères en majuscules.

Exemples d'utilisation

```
let message = "Hello World!";
let upper = message.toUpperCase(); // "HELLO WORLD!"
```

La méthode **toUpperCase()** permet de convertir une chaîne de caractères en majuscules.

- Cette méthode est utile pour normaliser les informations texte qui peuvent provenir de différentes sources.
- Elle est souvent utilisée pour comparer deux chaînes de caractères sans se soucier de la casse.

`trim()`

La méthode `trim()` supprime les **espaces**, les **tabulations** et les **retours à la ligne** au début et à la fin d'une chaîne.

```
let texte = "   Bonjour  \\n ";
console.log(texte.trim()); // "Bonjour"
```

Avant `trim()`	Après `trim()`
`" Bonjour \\n "`	`"Bonjour"`

La méthode `trim()` ne modifie pas la chaîne d'origine, elle retourne une nouvelle chaîne.

Syntaxe

```
let trimmedString = originalString.trim();
```

La méthode `trim()` enlève les espaces en début et fin de chaîne de caractères.

Exemples d'utilisation

```
let message = "  Hello, World!  ";
let trimmed = message.trim(); // "Hello, World!"
```

Utilisation de la méthode .trim() :

- Supprime les espaces inutiles au début et à la fin d'une chaîne de caractères

Il existe aussi d'autres méthodes pour manipuler les chaînes de caractères, comme .toUpperCase() et .toLowerCase().

`padStart()`

La méthode `padStart()` ajoute des caractères à gauche d'une chaîne jusqu'à atteindre une longueur déterminée.

Exemple :

```
let str = "5";
console.log(str.padStart(3, "0")); // Output: "005"

str = "123";
console.log(str.padStart(5, "0")); // Output: "00123"
```

Argument	Description
Longueur	La longueur totale souhaitée de la chaîne après padding
Caractère	Le caractère utilisé pour le padding (la valeur par défaut est l'espace)

La méthode `padEnd()` fonctionne de manière similaire, mais ajoute des caractères à droite de la chaîne.

Syntaxe

```
let paddedString = originalString.padStart(targetLength[, paddingCharacter]);
```

Cette fonction est utilisée pour compléter une chaîne avec un caractère spécifié jusqu'à atteindre une longueur cible. Si aucun caractère de remplissage n'est spécifié, il utilisera un espace par défaut.

Exemples d'utilisation

```
let number = "42";
let padded = number.padStart(6, "0"); // "000042"
```

La méthode **padStart** permet de compléter une chaîne de caractères avec une autre chaîne jusqu'à atteindre une longueur minimale. Le deuxième argument est la chaîne à répéter pour compléter la longueur.

`padEnd()`

La méthode `padEnd()` ajoute des caractères à **droite** d'une chaîne jusqu'à atteindre une **longueur déterminée**.

```
let chaine = "Bonjour";
let resultat = chaine.padEnd(10, "-");
console.log(resultat); // "Bonjour---"
```

Paramètre	Description
Longueur	La longueur finale à atteindre pour la chaîne.
Caractère	Caractère à utiliser pour compléter la chaîne.

Il existe une méthode similaire pour compléter la chaîne à gauche, elle s'appelle `padStart()`.

Syntaxe

```
let paddedString = originalString.padEnd(targetLength[, paddingCharacter]);
```

La méthode `padEnd()` permet de compléter une chaîne de caractères jusqu'à atteindre une longueur cible avec un caractère de remplissage spécifique.

Exemples d'utilisation

```
let message = "Hello";
let padded = message.padEnd(10, "!"); // "Hello!!!!!"
```

La méthode **padEnd** permet de compléter une chaîne de caractères jusqu'à atteindre une longueur spécifiée en ajoutant des caractères de remplissage à la fin de la chaîne.

Concaténation et interpolation de chaînes

Concaténation

La **concaténation** consiste à assembler deux **chaînes de caractères** bout à bout.

```
var str1 = 'Bonjour';
var str2 = ', ';
var str3 = 'monde!';

var result = str1 + str2 + str3;

console.log(result); // "Bonjour, monde!"
```

L'opérateur '+' est utilisé pour la concaténation de chaînes de caractères.

Syntaxe

Pour **concaténer** deux chaînes, vous pouvez utiliser l'opérateur `+`.

```
let chaine1 = "Bonjour, ";
let chaine2 = "comment ça va ?";
let message = chaine1 + chaine2;
```

Il existe plusieurs autres options pour concaténer les chaînes, la plus récente étant les **Template Strings** avec les backticks.

Exemples d'utilisation

```
let nom = "Dupont";
let prenom = "Jean";
let nomComplet = prenom + " " + nom;
```

Cet exemple montre comment concaténer des chaînes de caractères pour créer une chaîne nomComplet.

Interpolation

L'**interpolation** permet d'insérer des **variables** directement dans une **chaîne de caractères**.

```
const nom = "Jane";
const age = 30;

const message = `Bonjour, je m'appelle ${nom} et j'ai ${age} ans.`;
console.log(message); // "Bonjour, je m'appelle Jane et j'ai 30 ans."
```

Utilisation des backticks (`)

Pour utiliser l'**interpolation**, il faut utiliser les **backticks** pour délimiter la chaîne :

```
let age = 25;
let texte = `J'ai ${age} ans.`;
```

Les backticks permettent d'inclure des expressions JavaScript directement dans les chaînes de caractères.

Exemples d'utilisation

```
let a = 5;
let b = 3;
```

```
let somme = a + b;
let message = `La somme de ${a} et ${b} est ${somme}.`;
```

Cet exemple démontre une concaténation de chaînes de caractères **)** pour afficher dynamiquement les valeurs dans un message.

Comparaison de chaînes

Ordre lexicographique

Les **chaînes de caractères** sont comparées en fonction de leur **ordre lexicographique** (ordre du dictionnaire).

```
console.log("apple" < "banana"); // true
console.log("apple" > "banana"); // false
```

L'ordre lexicographique est basé sur la valeur Unicode des caractères.

Explication

L'**ordre lexicographique** dépend du **code Unicode** des caractères.

```
console.log("A" > "a"); // false
console.log("Z" < "a"); // true
```

Exemples d'utilisation

```
let a = "antoine";
let b = "bernard";

if (a < b) {
  console.log(`${a} vient avant ${b}`);
} else {
  console.log(`${b} vient avant ${a}`);
}
```

Méthode `localeCompare`

La méthode `**localeCompare**` permet de comparer les chaînes en tenant compte des spécificités **linguistiques**.

```
var str1 = "avion";
var str2 = "bateau";

console.log(str1.localeCompare(str2)); // Renvoie -1 car "avion" vient avant "bateau" dans
l'ordre lexicographique

console.log(str2.localeCompare(str1)); // Renvoie 1 car "bateau" vient après "avion" dans l'o
rdre lexicographique

console.log(str1.localeCompare(str1)); // Renvoie 0 car les deux chaînes sont identiques
```

Cette méthode est utile pour trier des tableaux de chaînes de caractères dans l'ordre alphabétique, en tenant compte des accents et d'autres spécificités linguistiques.

Syntaxe

```
let resultat = str1.localeCompare(str2);
```

- `resultat` :
  - **1** si `str1` vient avant `str2`
  - **1** si `str1` vient après `str2`
  - **0** si les chaînes sont égales

Cette méthode est utile pour la comparaison et le tri des chaînes de caractères en respectant les spécificités des différentes langues.

Exemples d'utilisation

```
let a = "éléphant";
let b = "écureuil";

let resultat = a.localeCompare(b);

if (resultat < 0) {
  console.log(`${a} vient avant ${b}`);
} else if (resultat > 0) {
  console.log(`${b} vient avant ${a}`);
} else {
  console.log(`${a} et ${b} sont égaux`);
}
```

Cet exemple montre comment utiliser `localeCompare` pour comparer deux chaînes de caractères en tenant compte des règles de tri locales.

Chaînes et Unicode

Caractères Unicode

Les **caractères Unicode** sont des symboles abstraits qui peuvent être représentés dans une chaîne de caractères en **JavaScript**.

- Les codes Unicode se situent entre U+0000 et U+10FFFF.
- Utilisez `\\u{codepoint}` pour représenter un caractère Unicode dans une chaîne de caractères.

```
  let chaîne = "Clé \\u{1F511}";
  console.log(chaîne); // Clé 🔑
```

Unicode est un système de codage de caractères qui permet de représenter des caractères de différentes langues et symboles, y compris des emojis.

Représentation avec escape sequences

Les caractères **Unicode** peuvent être représentés en utilisant des séquences d'échappement avec la syntaxe `\\u{CODE}` où `CODE` est le code hexadécimal du caractère.

```
let chaineUnicode = "Ceci est un exemple : \\u{1F600}";
console.log(chaineUnicode); // Ceci est un exemple : 😀
```

Les codes hexadécimaux peuvent être trouvés dans des références en ligne comme Unicode.org.
