JS

Objets: les bases

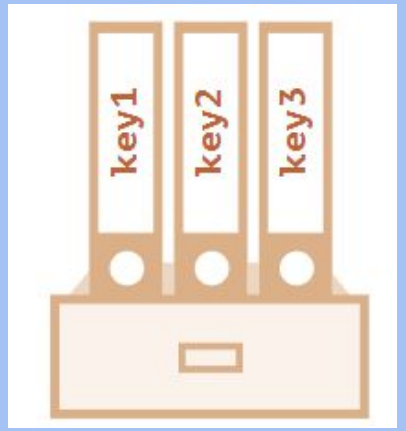
Objets

Les objets sont utilisés pour stocker des collections de données variées et d'entités plus complexes.

En JavaScript, les objets pénètrent dans presque tous les aspects du langage.
Nous devons donc d'abord les comprendre avant d'aller plus loin.

Un objet peut être créé avec des accolades {...}, avec une liste optionnelle de propriétés. Une propriété est une paire "clé: valeur", dans laquelle la clé (key) est une chaîne de caractères (également appelée "nom de la propriété"), et la valeur (value) peut être n'importe quoi.

Nous pouvons imaginer un objet comme une armoire avec des fichiers signés.
Chaque donnée est stockée dans son fichier par la clé. Il est facile de trouver un fichier par son nom ou d'ajouter/supprimer un fichier.



Objets

Un objet vide (“armoire vide”) peut être créé en utilisant l’une des deux syntaxes suivantes :

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

Habituellement, les accolades {...} sont utilisées. Cette déclaration s’appelle un littéral objet (object literal).

Littéraux et propriétés

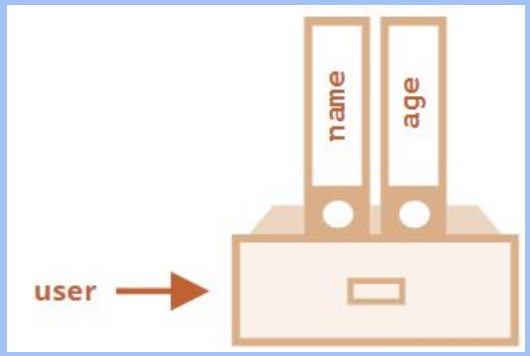
Nous pouvons immédiatement inclure certaines propriétés dans {...} sous forme de paires “clé: valeur” :

```
let user = {     // un objet
  name: "John",  // par clé "nom" valeur de stockage "John"
  age: 30        // par clé "age" valeur de stockage 30
};
```

Objets

Une propriété a une clé (également appelée “nom” ou “identifiant”) avant les deux points ":" et une valeur à sa droite. Dans l'objet **user**, il y a deux propriétés :

1. La première propriété porte le nom "name" et la valeur "John".
2. La seconde a le nom "age" et la valeur 30.



L'objet **user** résultant peut être imaginé comme une armoire avec deux fichiers intitulés “nom” et “âge”.

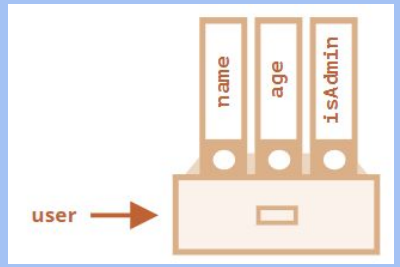
Nous pouvons ajouter, supprimer et lire des fichiers à tout moment.
Les valeurs de propriété sont accessibles à l'aide de la notation par points :

```
// récupère les valeurs de propriété de l'objet :
alert( user.name ); // John
alert( user.age );  // 30
```

Objets

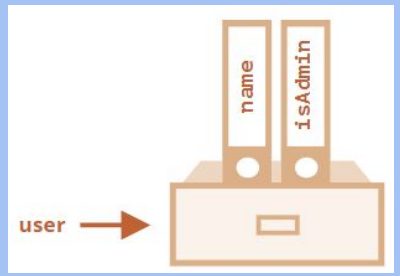
La valeur peut être de tout type. Ajoutons un booléen :

```
user.isAdmin = true;
```



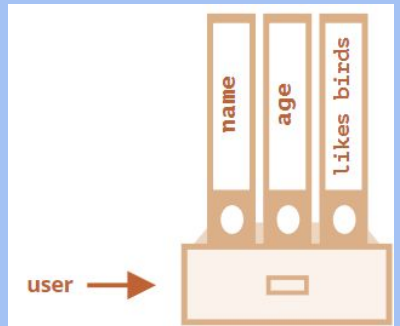
Pour supprimer une propriété, nous pouvons utiliser l'opérateur **delete** :e

```
delete user.age;
```



Nous pouvons également utiliser des noms de propriété *multi-mots*, mais ils doivent ensuite être entourés de "quotes" :

```
let user = {
  name: "John",
  age: 30,
  "likes birds": true // le nom de la propriété multi-mots
                      doit être entourée de quotes
};
```



Objets

La dernière propriété de la liste peut se terminer par une virgule :

```
let user = {
  name: "John",
  age: 30,
};
```

Cela s'appelle une virgule "trailing" ou "hanging". Elle facilite l'ajout/suppression/déplacement des propriétés, car toutes les lignes se ressemblent.

Crochets

Pour les propriétés multi-mots, l'accès par points ne fonctionne pas :

```
// cela donnerait une erreur de syntaxe
user.likes birds = true
```

JavaScript ne comprend pas cela. Il pense que nous adressons user.likes, ensuite il donne une erreur de syntaxe lorsqu'il rencontre des birds inattendus.

Le point nécessite que la clé soit un identificateur de variable valide. Cela implique qu'elle ne contient aucun espace, ne commence pas par un chiffre et n'inclut pas de caractères spéciaux ($ et _ sont autorisés).

Objets

Il existe une autre "notation entre crochets" qui fonctionne avec n'importe quelle chaîne :

```
let user = {};

// set
user["likes birds"] = true;

// get
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];
```

Maintenant tout va bien. Veuillez noter que la chaîne de caractères entre crochets est correctement entourée de quotes (tout type de guillemets fera l'affaire).
Les crochets fournissent également un moyen d'obtenir le nom de la propriété comme résultat de toute expression (par opposition à une chaîne de caractères littérale), semblable à une variable, comme ceci :

Objets

```
let key ="likes birds";// pareil que user["likes birds"] = true;
user[key] = true;
```

Ici, la variable key peut être calculée au moment de l'exécution ou dépendre de la saisie de l'utilisateur. Et ensuite, nous l'utilisons pour accéder à la propriété. Cela nous donne beaucoup de flexibilité.

Par exemple :

```
let user = {
  name: "John",
  age: 30
};

let key = prompt("What do you want to know about the user?", "name");

// accès par variable
alert( user[key] ); // John (si entré "name")
```

Objets

La notation par points ne peut pas être utilisée de la même manière :

```
let user = {
  name: "John",
  age: 30
};
let key = "name";

// accès par variable
alert( user.key ); // Undefined
```

Test d'existence de propriété, opérateur "in"

Une caractéristique notable des objets en JavaScript, par rapport à de nombreux autres langages, est qu'il est possible d'accéder à n'importe quelle propriété. Il n'y aura pas d'erreur si la propriété n'existe pas !

La lecture d'une propriété non existante renvoie simplement **undefined**. Nous pouvons donc facilement tester si la propriété existe :

Objets

La notation par points ne peut pas être utilisée de la même manière :

```
let user = {};

alert( user.noSuchProperty === undefined ); // true signifie "pas une telle propriété"
```

Il existe également un opérateur spécial "**in**" pour cela.

La syntaxe est :

```
"key" in object
```

Par exemple

```
let user = { name: "John", age: 30 };

alert("age" in user ); // true, user.age existe

alert("blabla" in user ); // false, user.blabla n'existe pas
```

Objets

Veuillez noter que sur le côté gauche de in, il doit y avoir un nom de propriété. C'est généralement une chaîne de caractères entre guillemets.
Si nous omettons les guillemets, cela signifie qu'une variable doit contenir le nom réel à tester. Par exemple :

```
let user = {age: 30 };
let key = "age";
alert( key in user ); // true, user.age existe
```

Pourquoi l'opérateur **in** existe-t-il ? N'est-ce pas suffisant de comparer avec **undefined** ?

Eh bien, la plupart du temps, la comparaison avec **undefined** fonctionne bien. Mais il y a un cas particulier quand il échoue, mais in fonctionne correctement.
C'est lorsque une propriété d'objet existe, mais qu'elle stocke **undefined** :

```
let obj = {test: undefined };
alert( obj.test ); // c'est indéfini, donc - pas une telle propriété
?...
alert( "test" in obj ); // ...true, la propriété existe
```

Objets

La boucle "for..in"

Pour parcourir toutes les clés d'un objet, il existe une forme spéciale de boucle : for..in. C'est une chose complètement différente de la construction for(;;) que nous avons étudiée auparavant.

La syntaxe :

```
for ( key in object ) {
  // exécute le corps pour chaque clé parmi les propriétés de l'objet
}
```

Par exemple, affichons toutes les propriétés de user :

```
let user = {name: "John", age: 30, isAdmin: true };

for (let key in user) {
  // keys
  alert( key );  // name, age, isAdmin
  // valeurs pour les clés
  alert( user[key] ); // John, 30, true
}
```

Notez que toutes les constructions "for" nous permettent de déclarer la variable en boucle à l'intérieur de la boucle, comme **let key** ici. En outre, nous pourrions utiliser un autre nom de variable ici au lieu de **key**. Par exemple, **for**(**let prop in obj**) est également largement utilisé.

Objets

Ordonné comme un objet

Les objets sont-ils ordonnés ? En d'autres termes, si nous parcourons un objet en boucle, obtenons-nous toutes les propriétés dans le même ordre où elles ont été ajoutées ? Pouvons-nous compter sur cela ?
La réponse courte est : "ordonné de manière spéciale" : les propriétés des entiers sont triées, les autres apparaissent dans l'ordre de création. Nous allons voir cela en détails.
Par exemple, considérons un objet avec les indicatifs de téléphone par pays :

```
let codes = {"49": "Germany", "41": "Switzerland","44": "Great Britain", // .., "1": "USA"};
for(let code in codes) {
  alert(code); // 1, 41, 44, 49
}
```

L'objet peut être utilisé pour suggérer une liste d'options à l'utilisateur. Si nous créons un site principalement pour le public allemand, nous voulons probablement que 49 soit le premier.
Mais si nous exécutons ce code, nous voyons une image totalement différente :

- USA (1) passe en premier
- puis Switzerland (41) et ainsi de suite.

Les indicatifs de téléphone sont classés par ordre croissant, **car ce sont des entiers**. Donc on voit 1, 41, 44, 49.
... Par contre, si les clés ne sont pas des entiers, **elles sont listées dans l'ordre de création.**

JS

Exercices

Les objets

Bonjour objet

Écrivez le code, une ligne pour chaque action :

- Créer un objet vide user.
- Ajoutez la propriété name avec la valeur John.
- Ajoutez la propriété surname avec la valeur Smith.
- Changer la valeur de name pour Pete.
- Supprimez la propriété name de l'objet.

Les objets

Bonjour objet - Réponse

```
let user = {};
user.name = "John";
user.surname = "Smith";
user.name = "Pete";
delete user.name;
```

JS

Objets: les bases

Les références d'objet et leur copie

Les objets

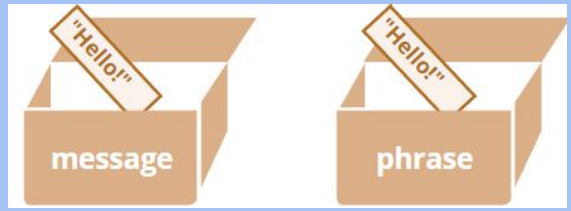
Les références d'objet et leur copie

Une des différences fondamentale des objets avec les primitives est que ceux-ci sont stockés et copiés "par référence", en opposition des valeurs primitives : strings, numbers, booleans, etc. – qui sont toujours copiés comme "valeur entière".

On comprendra plus facilement en regardant "sous le capot" ce qui se passe lorsque nous copions une valeur.
Commençons avec une primitive, comme une chaîne de caractères.
Ici nous assignons une copie de message dans phrase :

```
let message = "Hello!";
let phrase = message;
```

Il en résulte deux variables indépendantes, chacune stockant la chaîne de caractères "Hello!".



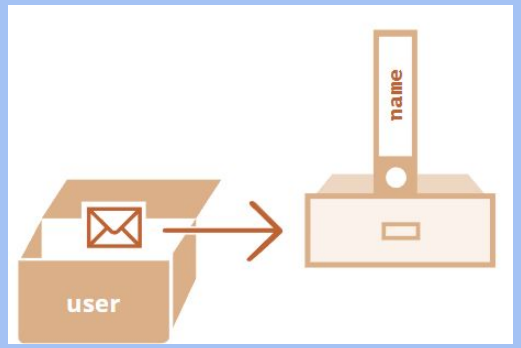
Un résultat plutôt évident n'est-ce pas ?
Les objets ne fonctionnent pas comme cela.

Les objets

Une variable assignée à un objet ne stocke pas l'objet lui-même, mais son "adresse en mémoire", en d'autres termes "une référence" à celui-ci.
Prenons un exemple d'une telle variable :

```
let user = {  name: "Pierre" };
```

Et ici comment elle est stockée en mémoire :
L'objet est stocké quelque part dans la mémoire (du coté droit de l'image), tandis que la variable user (du coté gauche) a une référence à celui-ci.



On peut imaginer la variable d'objet, ici user, comme une feuille de papier avec l'adresse de l'objet écrit dessus.

Lorque l'on réalise une action avec l'objet, par exemple récupérer la propriété user.name, le moteur de JavaScript regarde à l'adresse et réalise l'opération sur l'objet actuel.
Et voilà pourquoi cela est important !

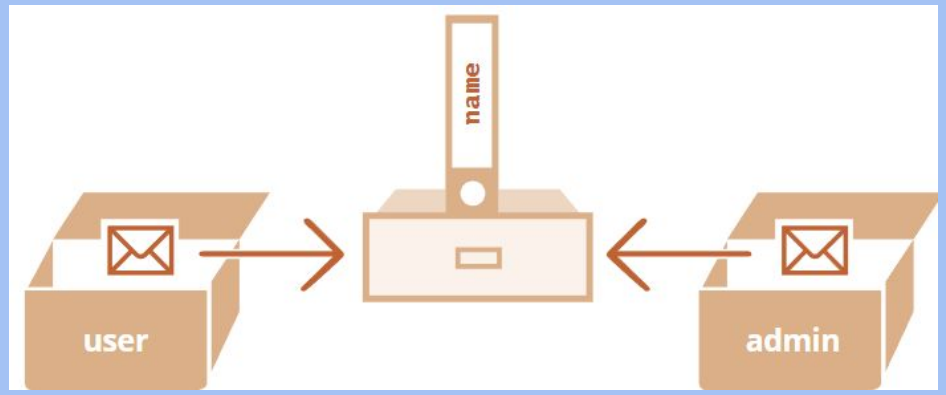
Lorsqu'une variable d'objet est copiée – la référence est copiée, l'objet lui-même n'est pas dupliqué.

Les objets

Par exemple :

```
let user = {  name: "Pierre" };
let admin = user; // copie la référence
```

Maintenant nous avons deux variables, chacune avec la référence vers le même objet :



Comme vous pouvez le voir, il n'y a toujours qu'un seul objet, mais maintenant avec deux variables qui le référence. On peut utiliser n'importe quelle variable pour accéder à l'objet et modifier son contenu :

Les objets

Par exemple :

```
let user = {  name: "Pierre" };
let admin = user;    // copie la référence
admin.name = 'Paul'; // changé par la référence "admin"
alert(user.name); // 'Paul', les changements sont visibles sur la référence "user"
```

C'est comme si nous avions une armoire avec deux clés et que nous en utilisions une (admin) pour y entrer et y apporter des modifications. Ensuite, si nous utilisons plus tard une autre clé (user), nous ouvrons toujours la même armoire et pouvons accéder au contenu modifié.

Les objets

Appel sans objet : this == undefined

Nous pouvons même appeler la fonction sans objet du tout :

```
function sayHi() {
  alert( this );
}
sayHi(); // undefined
```

Dans ce cas, **this** est **undefined** en mode strict.
Si nous essayons d'accéder à **this.name**, il y aura une erreur.

En mode non strict (si on oublie **use strict**), la valeur de **this** dans ce cas sera l'objet **global** (la fenêtre d'un navigateur, nous y reviendrons plus tard). *Ceci est un comportement historique que le mode strict corrige.*

Ce genre d'appel est généralement une erreur de programmation.
*Si il y a un **this** dans une fonction, il s'attend à être appelée dans un contexte d'objet.*

JS

Objets: les bases

Méthodes d'objet, "this"

Les objets

Méthodes d'objet, "this"

Les objets sont généralement créés pour représenter des entités du monde réel, comme des utilisateurs, des commandes, etc. :

```
let user = { name : "John", age :  30}
```

Et, dans le monde réel, un utilisateur peut agir : sélectionner un élément du panier, se connecter, se déconnecter, etc. Les actions sont représentées en JavaScript par des fonctions dans les propriétés.

Exemples de méthodes

Pour commencer, apprenons à user à dire bonjour :

```
let user = { name : "John", age :  30};
user.sayHi = function() {
  alert("Hello!");
};
user.sayHi(); // Hello!
```

Les objets

Ici, nous venons d'utiliser une fonction expression pour créer la fonction et l'affecter à la propriété *user.sayHi* de l'objet.
Ensuite, nous pouvons l'appeler comme user.sayHi(). L'utilisateur peut maintenant parler!
Une fonction qui est la propriété d'un objet s'appelle sa méthode.
Nous avons donc ici une méthode sayHi de l'objet user.

Bien sûr, nous pourrions utiliser une fonction pré-déclarée comme méthode, comme ceci :

```
let user = { name : "John", age :  30};
// d'abord, déclarer
function sayHi() {
  alert("Hello!");
};

// puis ajouter comme une méthode
user.sayHi = sayHi;
user.sayHi(); // Hello!
```

Les objets

Programmation orientée objet

Lorsque nous écrivons notre code en utilisant des objets pour représenter des entités, cela s'appelle une programmation orientée objet, en bref : "**POO**".
La programmation orientée objet est un élément important dans la conception du code

Méthode abrégée

Il existe une syntaxe plus courte pour les méthodes dans un littéral d'objet :

.

```
// ces objets font la même chose
user = {
  sayHi: function() {
    alert("Hello");
  }
};
// la méthode abrégée semble mieux, non ?
user = {
  sayHi() { // identique à "sayHi: function(){...}"
    alert("Hello");
  }
};
```

Les objets

Comme démontré, nous pouvons omettre "function" et simplement écrire sayHi().
A vrai dire, les notations ne sont pas totalement identiques. Il existe des différences subtiles liées à l'héritage d'objet (à couvrir plus tard), mais pour le moment, elles importent peu. Dans presque tous les cas, la syntaxe la plus courte est préférable.

"this" dans les méthodes

Il est courant qu'une méthode d'objet ait besoin d'accéder aux informations stockées dans l'objet pour effectuer son travail.
Par exemple, le code à l'intérieur de ***user.sayHi()*** peut nécessiter le nom de ***user***.

Pour accéder à l'objet, une méthode peut utiliser le mot-clé **this**.
La valeur de **this** est l'objet "avant le point", celui utilisé pour appeler la méthode.

Ici, lors de l'exécution de **user.sayHi(),** la valeur de this sera user.

Techniquement, il est également possible d'accéder à l'objet sans this, en le référençant via la variable externe :

```
let user = {
name:"Paul",
age:25,
sayHi(){
  alert( this.name );
};
user.sayHi(); // Paul !
```

Les objets

... Mais un tel code n'est pas fiable. Si nous décidons de copier **user** dans une autre variable, par exemple **admin = user** et écraser **user** avec quelque chose d'autre, il accédera au mauvais objet.

Cela est démontré ci-dessous :

```
let user = {
name:"Paul",
age:25,
sayHi(){
  alert( user.name ); // Mène à l'erreur
};
let admin = user;
user = null;  // écrase la valeur de user
admin.sayHi();// TypeError: Cannot read property 'name' of null
```

```
let user = {
name:"Paul",
age:25,
sayHi(){
  alert( user.name );
};
user.sayHi(); // Paul !
```

Si nous utilisions **this.name** au lieu de **user.name** dans l'alert, le code fonctionnerait !

Les objets

“this” n’est pas lié

En JavaScript, le mot clé **this** se comporte différemment de la plupart des autres langages de programmation.
Il peut être utilisé dans n’importe quelle fonction, même si ce n’est pas une méthode d’un objet.

Il n’y a pas d’erreur de syntaxe dans le code suivant :

```
function sayHi() {
  alert( this.name );
}
```

La valeur de **this** est évaluée pendant l’exécution, en fonction du contexte.

Les objets

Par exemple, ici la même fonction est assignée à deux objets différents et a un "this" différent dans les appels :

```
let user = { name: "Jean" };
let admin = { name: "Admin" };
function sayHi() {
  alert( this.name );
}

// utiliser la même fonction dans deux objets
user.f = sayHi;
admin.f = sayHi;

// ces appels ont un this différent
// "this" à l'intérieur de la fonction est l'objet "avant le point"
user.f(); // Jean  (this == user)
admin.f(); // Admin  (this == admin)
```

La règle est simple : si obj.f() est appelé, alors this est obj pendant l'appel de f. C'est donc l'user ou l'admin dans l'exemple ci-dessus.

JS

Exercices

Les objets

Créer une "calculatrice"

Créez un objet **calculator** avec trois méthodes :

1. **read()** demande deux valeurs et les enregistre en tant que propriétés d'objet avec les noms **a** et **b**.
2. **sum()** renvoie la somme des valeurs sauvegardées.
3. **mul()** multiplie les valeurs sauvegardées et renvoie le résultat.

```
let calculator = {
  // ... votre code ...
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

Les objets

La calculatrice - Réponse

```
let calculator = {
 sum() {
    return this.a + this.b;
  },
  mul() {
    return this.a * this.b;
  },
  read() {
    this.a = +prompt('a?', 0);
    this.b = +prompt('b?', 0);
  }
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

JS

Objets: les bases

Le constructeur, l'opérateur "new"

Le constructeur, l'opérateur "new"

La syntaxe normale {...} permet de créer un seul objet. Mais souvent, nous devons créer de nombreux objets similaires, tels que plusieurs utilisateurs ou éléments de menu, etc.
Cela peut être fait en utilisant les fonctions constructeur et l'opérateur "**new**".

La function constructeur

Les fonctions constructeur sont techniquement des fonctions habituelles. Il existe cependant deux conventions :

1. Elles sont nommées avec **une lettre majuscule en premier**.
2. Elles ne devraient être executées qu'avec l'opérateur "**new**".

Par exemple :

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

Le constructeur, l'opérateur "new"

Quand une fonction est exécutée avec **new**, elle effectue les étapes suivantes :

Un nouvel objet vide est créé et affecté à **this**.
Le corps de la fonction est exécuté.
Habituellement, il modifie **this**, y ajoutant de nouvelles propriétés.
La valeur de this est retournée.

En d'autres termes, **new User(...)** fait quelque chose comme ça :

```
function User(name) {
  // this = {};  (implicitement)
  // ajoute des propriétés à this
  this.name = name;
  this.isAdmin = false;

  // return this;  (implicitement)
}
```

Donc **let user = new User("Jack")** donne le même résultat que :

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

Le constructeur, l'opérateur "new"

Les méthodes dans les constructeurs

L'utilisation de fonctions de constructeur pour créer des objets offre une grande flexibilité.

La fonction constructeur peut avoir des paramètres qui définissent comment construire l'objet et ce qu'il doit y mettre.

Bien sûr, nous pouvons ajouter à this non seulement des propriétés, mais également des méthodes.

Par exemple, **new User(name)** ci-contre créer un objet avec le **name** donné et la méthode **sayHi** :

```
function User(name) {
  this.name = name;
  this.sayHi = function() {
    alert( "Mon nom est: " + this.name );
  };
}

let john = new User("Kevin");

john.sayHi(); // Mon nom est: Kevin

/*
kevin= {
   name: "kevin",
   sayHi: function() { ... }
}
*/
```

Le constructeur, l'opérateur "new"

Par exemple, ici **return** remplace **this** en retournant un objet :

```
function bigUser() {
  this.name = "Robert";
  return {name:"Yves"};// <-- retourne cet objet
}

alert( new bigUser().name ); // Yves, comme objet
```

Et voici un exemple avec un **return** vide (ou nous pourrions placer une primitive après, peu importe) :

```
function smallUser() {
  this.name = "Robert";
  return;
}

alert( new smallUser().name ); // Robert
```

JS

Types de données

Méthodes des primitives

JavaScript nous permet de travailler avec des primitives (chaînes de caractères, nombres, etc.) comme s'il s'agissait d'objets. Ils prévoient également des méthodes pour les appeler en tant que tel. Nous étudierons cela très bientôt, mais nous verrons d'abord comment cela fonctionne car, bien entendu, les primitives ne sont pas des objets (et nous allons rendre cela plus clair).

Examinons les principales différences entre primitives et objets.

Une primitive

- Est une valeur de type primitif.
- Il existe 7 types primitifs : **string, number, bigint, boolean, symbol, null** et **undefined**.

Un objet

- Est capable de stocker plusieurs valeurs en tant que propriétés.
- Peut être créé avec {}, par exemple : {name: "John", age: 30}. Il existe d'autres types d'objets en JavaScript. Les fonctions, par exemple, sont des objets.

Méthodes des primitives

L'une des meilleurs choses à propos des objets est que nous pouvons stocker une fonction en tant que l'une de ses propriétés.
Nous avons donc crée un objet jean avec la méthode sayHI.

```
let jean = {
  name: "Jean",
  sayHi: function() {
    alert("Bonjour !");
  }
};

jean.sayHi(); // Bonjour !
```

De nombreux objets intégrés existent déjà, tels que ceux qui fonctionnent avec des dates, des erreurs, des éléments HTML, etc. Ils ont des propriétés et des méthodes différente.
Mais, ces fonctionnalités ont un coût !
Les objets sont "plus lourds" que les primitives. Ils ont besoin de ressources supplémentaires pour soutenir le mécanisme interne.

Une primitive en tant qu'objet
Voici le paradoxe auquel est confronté le créateur de JavaScript :
Il y a beaucoup de choses que l'on voudrait faire avec une primitive telle qu'une chaîne de caractères ou un nombre. Ce serait génial d'y avoir accès avec des méthodes.
Les primitives doivent être aussi rapides et légères que possible.

Méthodes des primitives

La solution semble peu commode, mais la voici :

- Les primitives sont toujours primitives. Une seule valeur, au choix.
- Le langage permet d'accéder aux méthodes et aux propriétés des chaînes de caractères, des nombres, des booléens et des symboles.
- Pour que cela fonctionne, un "wrapper d'objet" (conteneur) spécial est créé pour fournir la fonctionnalité supplémentaire, puis il est détruit.

Les "wrapper d'objets" (conteneurs) sont différents pour chaque type de primitive et sont appelés String, Number, Boolean et Symbol. Ainsi, ils fournissent différents ensembles de méthodes.

Par exemple, il existe une méthode de string **str.toUpperCase()** qui renvoie une chaîne de caractères **str** en majuscule.

Voici comment ça fonctionne:

```
let str = "Hello";
alert( str.toUpperCase() ); // HELLO
```

Méthodes des primitives

Simple, non? Voici ce qui se passe réellement dans **str.toUpperCase()**:

- La chaîne de caractères **str** est une primitive. Ainsi, au moment d'accéder à sa propriété, un objet spécial est crée, qui connaît la valeur de la chaîne de caractères et possède des méthodes utiles, comme **toUpperCase().**
- Cette méthode s'exécute et retourne une nouvelle chaîne de caractères (indiquée par alert).
- L'objet spécial est détruit, laissant le primitif str seul.

Les primitives peuvent donc fournir des méthodes, mais elles restent légères.

Le moteur JavaScript optimise fortement ce processus. Il peut même ignorer la création de l'objet supplémentaire. Mais il doit toujours adhérer à la spécification et se comporter comme s'il en crée un.

Un nombre a ses propres méthodes, par exemple, **toFixed(n)** arrondit le nombre à la précision indiquée :

```
let n = 1.23456;
alert( n.toFixed(2) ); // 1.23
```

Nombres

En JavaScript moderne, il existe deux types de nombres :

1. Les nombres standards en JavaScript sont stockés au format 64 bits IEEE-754, également connu sous le nom de "nombres à virgule flottante double précision". Ce sont des chiffres que nous utilisons le plus souvent, et nous en parlerons dans ce chapitre.
2. Les nombres BigInt pour représenter des entiers de longueur arbitraire. Ils sont parfois nécessaires, car un nombre régulier ,number, ne peut pas dépasser de manière précise 2^{53} ou être inférieur à -2^{53}.

Nous allons donc parler ici des nombres réguliers.

Plus de façons d'écrire un nombre
Imaginez que nous ayons besoin d'écrire 1 milliard. Le moyen évident est :

```
let milliard = 1000000000;
```

Nous pouvons également utiliser l'underscore _ comme séparateur :

```
let milliard = 1_000_000_000;
```

Nombres

Ici, l'underscore _ joue le rôle de "sucre syntaxique", il rend le nombre plus lisible. Le moteur JavaScript ignore simplement _ entre les chiffres, donc c'est exactement le même milliard que ci-dessus.

Dans la vraie vie cependant, nous essayons d'éviter d'écrire de longues séquences de zéros. Nous sommes trop paresseux pour ça. Nous essaierons d'écrire quelque chose comme "1 milliard" pour un milliard ou "7,3 milliards" pour 7 milliards 300 millions. La même chose est vraie pour la plupart des grands nombres.

En JavaScript, nous pouvons raccourcir un nombre en y ajoutant la lettre "e" et en spécifiant le nombre de zéros :

```
let milliard = 1e9;  // 1 milliard, littéralement : 1 suivi de 9 zéros
alert( 7.3e9 );  // 7.3 milliards (pareil que 7300000000 ou 7_300_000_000)
```

En d'autres termes, e multiplie le nombre par 1 suivi du nombre de zéros spécifié.

```
1e3 === 1 * 1000 // e3 signifie *1000
1.23e6 === 1.23 * 1000000 // e6 signifie *1000000
```

Nombres

Maintenant, écrivons quelque chose de très petit. Disons, 1 microseconde (un millionième de seconde) :

```
let ms = 0.000001;
```

Comme avant, l'utilisation de "e" peut nous aider. Si nous voulons éviter d'écrire les zéros explicitement, nous pourrions dire la même chose comme :

```
let ms = 1e-6;// cinq zéros à gauche de 1
```

Si nous comptons les zéros dans 0.000001, il y en a 6. Donc logiquement, c'est 1e-6.
En d'autres termes, un nombre négatif après "e" signifie une division par 1 suivi du nombre spécifié de zéros :

```
1e-3 === 1 / 1000 // -3 divise par 1 avec 3 zéros soit 0.001
1.23e-6 === 1.23 / 1000000 // -6 divise par 1 avec  6 zéros 0.00000123
```

Nombres hexadécimaux, binaires et octaux

Les nombres Hexadécimaux sont souvent utilisés en JavaScript pour représenter des couleurs, encoder des caractères et pour beaucoup d'autres choses. Alors, naturellement, il existe un moyen plus court de les écrire : **0x puis le nombre.**

Nombres

Pax exemple

```
alert( 0xff ); // 255
alert( 0xFF ); // 255 (même résultat car la casse n'a pas d'importance)
```

Les systèmes numériques binaires et octaux sont rarement utilisés, mais sont également supportés avec les préfixes 0b et 0o :

```
let a = 0b11111111; // forme binaire de 255
let b = 0o377;      // forme octale de 255
alert( a == b );    // true, car a et b sont le même nombre, 255
```

Cependant ça ne fonctionne qu'avec ces 3 systèmes de numération. Pour les autres systèmes numérique, nous devrions utiliser la fonction **parseInt** (que nous verrons plus loin dans ce chapitre).

La méthode toString(base)

La méthode **num.toString(base)** retourne une chaîne de caractères représentant **num** dans le système numérique de la base donnée.

Par exemple :

```
let num = 255;
alert(num.toString(16)); // ff
alert(); // 1111111
```

Nombres

La base peut varier de 2 à 36. Par défaut, il s'agit de 10.
Les cas d'utilisation courants sont :

- **base=16** est utilisé pour les couleurs hexadécimales, les encodages de caractères, etc. Les chiffres peuvent être 0..9 ou A..F.
- **base=2** est principalement utilisé pour le débogage d'opérations binaires, les chiffres pouvant être 0 ou 1.
- **base=36** est le maximum, les chiffres peuvent être 0..9 ou A..Z. L'alphabet latin entier est utilisé pour représenter un nombre. Un cas amusant mais utile pour la base 36 consiste à transformer un identifiant numérique long en quelque chose de plus court, par exemple pour créer une URL courte. On peut simplement le représenter dans le système numérique avec base 36 :

```
alert(123456.toString(36)); // 2n9c
```

Arrondir

Arrondir est l'une des opérations les plus utilisées pour travailler avec des nombres. Il existe plusieurs fonctions intégrées pour arrondir :

- **Math.floor** : Arrondis vers le bas : 3.1 devient 3, et -1.1 devient -2.
- **Math.ceil :** Arrondis ver le haut : 3.1 devient 4, et -1.1 devient -1.

Nombres

- **Math.round** : Arrondit à l'entier le plus proche : 3,1 devient 3, 3,6 devient 4 et pour le cas du milieu, 3,5 est également arrondit à 4.
- **Math.trunc** : Supprime tout ce qui suit le point décimal : 3.1 devient 3, -1.1 devient -1.

Voici le tableau pour résumer les différences entre eux :

	Math.floor	Math.ceil	Math.round	Math.trunc
3.1	3	4	3	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

Ces fonctions couvrent toutes les manières possibles de traiter la partie décimale d'un nombre. Mais que se passe-t-il si nous voulons arrondir le nombre à un certain chiffre après la virgule ?

Par exemple, nous avons 1.2345 et voulons l'arrondir à 2 chiffres, pour obtenir seulement 1.23.

Il y a deux façons de le faire:

Nombres

1. **Multiplier et diviser.**

Par exemple, pour arrondir le nombre au 2ème chiffre après la décimale, nous pouvons multiplier le nombre par "100", appeler la fonction d'arrondi puis le diviser.

```
let num = 1.23456;
alert( Math.round(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

2. La méthode **toFixed(n)** arrondit le nombre à n chiffres après le point et renvoie une chaîne de caractères du résultat.

```
let num = 12.36;
alert( num.toFixed(1) ); // "12.4"
```

Veuillez noter que le résultat de **toFixed** est une chaîne de caractères. Si la partie décimale est plus courte qu'indiquée, des zéros sont ajoutés à la fin :

```
let num = 12.34;
alert( num.toFixed(5) ); // "12.34000"
```

Nous pouvons le convertir en un nombre en utilisant le plus unaire **+** ou un appel **Number()** : **+num.toFixed(5).**

Nombres

isFinite et isNaN

- Infinite (et -Infinite) sont des valeurs numériques spéciales qui sont supérieures (inférieure) à tout.
- NaN représente une erreur.

Ils appartiennent au type number, mais ne sont pas des numéros "normaux". Il existe donc des fonctions spéciales pour les vérifier :

- **isNaN(valeur)** convertit son argument en un nombre et teste ensuite s'il est **NaN** :

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
```

Mais avons-nous besoin de cette fonction ? Ne pouvons-nous pas simplement utiliser la comparaison === NaN ? Malheureusement non. La valeur NaN est unique en ce sens qu'elle ne vaut rien, y compris elle-même :

```
alert(NaN === NaN ); // false
```

Nous pouvons le convertir en un nombre en utilisant le plus unaire **+** ou un appel **Number()** : **+num.toFixed(5).**

Nombres

- **isFinite(valeur)** convertit son argument en un nombre et renvoie true s'il s'agit d'un nombre régulier, pas de NaN / Infinity / -Infinity :

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, car c'est une valeur non régulière: NaN
alert( isFinite( Infinity ) ); // false, car c'est une valeur non régulière
```

Parfois, **isFinite** est utilisé pour valider si une valeur de chaîne de caractères est un nombre régulier

```
let num = +prompt("Entrer un nombre","");
// sera vrai, sauf si vous entrez Infinity, -Infinity ou NaN
alert( isFinite( num ) );
```

Veuillez noter qu'une chaîne de caractères vide ou une chaîne de caractères contenant seulement des espaces est traitée comme **0** dans toutes les fonctions numérique, y compris **isFinite**.

Nombres

parseInt et parseFloat

La conversion numérique à l'aide d'un plus + ou Number() est strict. Si une valeur n'est pas exactement un nombre, elle échoue :

```
alert( +"100px" ); // NaN
```

La seule exception concerne les espaces au début ou à la fin de la chaîne de caractères, car ils sont ignorés. Mais dans la vraie vie, nous avons souvent des valeurs en unités, comme "100px" ou "12pt" en CSS. En outre, dans de nombreux pays, le symbole monétaire se situe après le montant. Nous avons donc "19€" et souhaitons en extraire une valeur numérique.

C'est à quoi servent **parseInt** et **parseFloat**.

Ils "lisent" un nombre d'une chaîne jusqu'à ce qu'ils ne puissent plus. En cas d'erreur, le numéro rassemblé est renvoyé. La fonction **parseInt** renvoie un entier, tandis que **parseFloat** renvoie un nombre à virgule :

```
alert( parseInt('100px') );  // 100
alert( parseFloat('12.5rem') ); // 12.5
alert( parseInt('12.3') ); // 12, seule la partie entière est renvoyée
alert( parseFloat('12.3.4') ); ); // 12.3, le deuxième point arrête la lecture
```

Nombres

Il y a des situations où parseInt / parseFloat retournera NaN. Cela arrive quand on ne peut lire aucun chiffre :

```
alert( parseInt('a123') ); // NaN, le premier symbole arrête le processus
```

Le second argument de parseInt(str, radix)

La fonction parseInt() a un second paramètre optionnel. Il spécifie la base du système numérique, ce qui permet à parseInt d'analyser également les chaînes de nombres hexadécimaux, binaires, etc. :

```
alert( parseInt('0xff', 16) );  // 255
alert( parseFloat('ff', 16) ); // 255, sans 0x cela fonctionne également
alert( parseInt('2n9c'), 36 ); // 12456
```

Autres fonctions mathématiques

JavaScript a un objet Math intégré qui contient une petite bibliothèque de fonctions et de constantes mathématiques.

Nombres

Quelques exemples :

Math.random()

Retourne un nombre aléatoire de 0 à 1 (1 non compris).

```
alert( Math.random() ); //...(tout nombre aléatoire)
```

Math.max(a, b, c...) et Math.min(a, b, c...)

Renvoie le plus grand et le plus petit d'un nombre arbitraire d'arguments.

```
alert( Math.max(3,5,-10,0,2) ); // 5
alert( Math.min(1,2) ); // 1
```

Math.pow(n, power)

Renvoie n élevé à la puissance power donnée.

```
alert( Math.pow(2,10) ); // 2 puissance 10 = 1024
```

Strings

En JavaScript, les données de type texte sont stockées sous forme de chaînes de caractères.
Il n'y a pas de type séparé pour un seul caractère.
Le format interne des chaînes de caractères est toujours UTF-16, il n'est pas lié au codage de la page.

Quotes

Rappelons les types de quotes.
Les chaînes de caractères peuvent être placées entre guillemets simples, doubles ou backticks :

```
let single = 'single-quoted';
let double = 'double-quoted';
let backticks = `backticks`;
```

Les guillemets simples et doubles sont essentiellement les mêmes. Les backticks nous permettent toutefois d'incorporer n'importe quelle expression dans la chaîne de caractères, en l'enveloppant dans ${...} :

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

Strings

L'utilisation des backticks présente également l'avantage de permettre à une chaîne de caractères de couvrir plusieurs lignes :

```
let guestList = `Guests:
 * John
 * Pete
 * Mary
`;

alert(guestList);
// une liste d'invités sur
plusieurs lignes
```

Ça a l'air naturel, non? Mais les guillemets simples ou doubles ne fonctionnent pas de cette façon.
Si nous les utilisons et essayons d'utiliser plusieurs lignes, il y aura une erreur :

```
let guestList = "Guests: // Error: Unexpected token ILLEGAL
 * John";
```

Les guillemets simples et doubles proviennent d'anciens temps de la création linguistique lorsque la nécessité de chaînes multilignes n'était pas prise en compte. Les backticks sont apparus beaucoup plus tard et sont donc plus polyvalents.

Caractères spéciaux
Il est encore possible de créer des chaînes de caractères multilignes avec des guillemets simples et doubles en utilisant un "caractère de nouvelle ligne", écrit comme ceci \n, qui spécifie un saut de ligne :

Strings

```
let guestList = "Guests:\n * John\n * Pete\n * Mary";
alert(guestList); // une liste d'invités sur plusieurs lignes
```

Comme exemple plus simple, ces deux lignes sont égales, juste écrites différemment :

```
let str1 = "Hello\nWorld"; // deux lignes utilisant un "symbole de nouvelle ligne"
// deux lignes utilisant une nouvelle ligne normale et des backticks
let str2 = `Hello
World`;
alert( str1 == str2 ); // true
```

Il existe d'autres caractères "spéciaux" moins courants.

Voici la liste complète :

Car.	Description
\n	Nouvelle ligne
\r	Dans les fichiers texte Windows, une combinaison de deux caractères \r\n représente une nouvelle ligne, tandis que sur un système d'exploitation non Windows, il s'agit simplement de \n. C'est pour des raisons historiques, la plupart des logiciels Windows comprennent également \n.
\', \"	Quotes
\\	Backslash
\t	Tab

Strings

Longueur de chaîne de caractères

La propriété **length** indique la longueur de la chaîne de caractères :

```
alert( `Mr\n`.length ); // 3
```

Notez que \n est un seul caractère "spécial", la longueur est donc bien 3.

length est une propriété

Les personnes ayant des connaissances dans d'autres langages peuvent parfois commettre des erreurs en l'appelant *str.length()* au lieu de **str.length**. Cela ne fonctionnera pas.

Veuillez noter que **str.length** est une propriété numérique et non une fonction. Il n'est pas nécessaire d'ajouter des parenthèses après. Pas *.length()*, mais **.length**.

Accéder aux caractères

Pour obtenir un caractère à la position pos, utilisez des crochets [pos] ou appelez la méthode str.at(pos). Le premier caractère commence à la position zéro :

```
let str = `Hello`;
// le premier caractère
alert( str[0] ); // H
alert( str.at(0) ); // H
```

```
let str = `Hello`;
// le dernier caractère
alert( str[str.length - 1] ); // o
alert( str.at(-1) ); // o
```

Strings

Comme vous pouvez le voir, la méthode **.at(pos)** a l'avantage de permettre une position négative.
Si pos est négatif, alors il est compté à partir de la fin de la chaîne de caractères.
Donc **.at(-1) s**ignifie le dernier caractère, et **.at(-2)** est celui qui le précède, etc.
Les crochets renvoient toujours ***undefined*** *pour les index négatifs, par exemple :*

```
let str = `Hello`;
// le dernier caractère
alert( str[-2] ); // undefined
alert( str.at(-2) ); // l
```

Nous pouvons également parcourir les caractères en utilisant un for..of :

```
for ( let char of "Hello" ){
   alert( char  ); // H,e,l,l,o (char devient "H", ensuite "e", ensuite "l", etc.)
}
```

Les chaînes de caractères sont immuables
Les chaînes de caractères ne peuvent pas être changées en JavaScript. Il est impossible de modifier un caractère. Essayons de démontrer que cela ne fonctionne pas :

Strings

```
let str = 'Hi';
str[0] = 'h'; // Erreur
alert( str[0] ); // ne fonctionne pas
```

La solution habituelle consiste à créer une nouvelle chaîne et à l'affecter à str au lieu de l'ancienne.
Par exemple :

```
let str = 'Hi';
str = 'h' + str[1]; // Remplace la chaine de caractères
alert( str ); // hi
```

Modifier la casse

Les méthodes **toLowerCase()** et **toUpperCase()** modifient la casse :

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

Ou, si nous voulons un seul caractère minuscule :

```
alert( 'Interface'[0].toUpperCase() ); // 'i'
```

Strings

Rechercher un substring (partie de la chaîne de caractères)

Il existe plusieurs façons de rechercher une partie d'une chaîne de caractères.

str.indexOf

La première méthode est str.indexOf(substr, pos).
Il cherche le substr dans str, en partant de la position donnée pos, et retourne la position où la correspondance a été trouvée ou -1 si rien ne peut être trouvé.
Par exemple :

```
let str = 'Widget avec id';
alert( str.indexOf('Widget') ); // 0, parce que 'Widget' est trouvé au début
alert( str.indexOf('widget') ); // -1, pas trouvé, la recherche est sensible à la casse
alert( str.indexOf('id') ); // 1, "id" est trouvé à la position 1 (..idget avec id)
```

Le second paramètre optionnel nous permet de rechercher à partir de la position donnée.
Par exemple, la première occurrence de "id" est à la position 1. Pour rechercher l'occurrence suivante, commençons la recherche à partir de la position 2 :

```
alert( str.indexOf('id', 2) ); // 12, "id" est trouvé à la position 12 (..avec id)
```

Strings

Si toutes les occurrences nous intéressent, nous pouvons exécuter indexOf dans une boucle. Chaque nouvel appel est passé avec la position après le match précédent :

```
let str = "Aussi rusé qu'un renard, aussi rapide qu'un léopard";
let target = 'qu'; // cherchons les caractères 'qu'
let pos = 0; // On commence à la position 0 premier caractère
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;
  alert( `Trouvé à la position ${foundPos}` );
  pos = foundPos + 1; // continue la recherche à partir de la position suivante
}
```

Le même algorithme peut être raccourci :

```
let str = "Aussi rusé qu'un renard, aussi rapide qu'un léopard";
let target = 'qu'; // cherchons les caractères 'qu'
let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}
```

Strings

includes, startsWith, endsWith

La méthode plus moderne **str.includes(substr, pos)** retourne true/false en fonction de si str contient substr.
C'est le bon choix si nous devons tester la présence, mais n'avons pas besoin de sa position :

```
alert( "Widget avec id".includes("Widget") ); // true
alert( "Widget avec id".includes("Test") ); // false
```

Le deuxième argument optionnel de **str.includes** est la position de départ de la recherche :

```
alert( "Widget".includes("id") ); // true
alert( "Widget".includes("id", 3) ); // false, à partir de la position 3, il n'y a pas de "id"
```

Les méthodes **str.startsWith** et **str.endsWith** font exactement ce qu'elle disent :

```
alert( "Widget".startsWith("Wid") ); // true, "Widget" démarre avec "Wid"
alert( "Widget".endsWith("get") ); // true, "Widget" fini avec "get"
```

Strings

Obtenir un substring (sous-chaîne de caractères)

Il existe 3 méthodes en JavaScript pour obtenir une "sous chaîne" : **substring**, **substr** et **slice**.

str.slice(start [, end])

Renvoie la partie de la chaîne de caractères de start jusqu'à (sans l'inclure) end. Par exemple :

```
let str = "stringify";
alert( str.slice(0, 5) ); // 'strin', le substring de 0 à 5 (sans inclure 5)
alert( str.slice(0, 1) ); // 's', de 0 à 1, mais sans inclure 1, donc uniquement le
caractère à l'index 0
```

S'il n'y a pas de second argument, slice va jusqu'à la fin de la chaîne de caractères :

```
let str = "stringify";
alert( str.slice(2) ); // 'ringify', à partir de la 2e position jusqu'à la fin
```

Des valeurs négatives pour start/end sont également possibles. Elles veulent dire que la position est comptée à partir de la fin de la chaîne de caractères :

```
let str = "stringify";
alert( str.slice(-4,-1) ); // 'gif'
```

Strings

str.substring(start [, end])

Renvoie la partie de la chaîne de caractères entre start et end (end non inclus).

C'est presque la même chose que slice, mais cela permet à start d'être supérieur à end (dans ce cas, il échange simplement les valeurs start et end). Par exemple :

```
let str = "stringify";
// ce sont les mêmes pour substring
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"
// ...mais pas pour slice :
alert( str.slice(2, 6) ); // "ring" (le même résultat)
alert( str.slice(6, 2) ); // "" (une chaîne de caractères vide)
```

Les arguments négatifs ne sont pas supportés (contrairement à slice), ils sont traités comme 0.

str.substr(start [, length])

Renvoie la partie de la chaîne de caractères à partir de start, avec le length (longueur) donné.

Contrairement aux méthodes précédentes, celle-ci nous permet de spécifier la longueur length au lieu de la position finale :

Strings

```
let str = "stringify";
alert( str.substring(2, 4) ); // 'ring', à partir de la 2ème position on obtient 4 caractères
```

Le premier argument peut être négatif, pour compter à partir de la fin :

```
let str = "stringify";
alert( str.substring(-4,2) ); // // 'gi', à partir de la 4ème position on obtient 2 caractères
```

Récapitulons ces méthodes pour éviter toute confusion :

méthodes	**séléction ...**	**valeurs negatives**
slice(start, end)	de start à end (n'inclue pas end)	permet les négatifs
substring(start, end)	entre start et end	les valeurs négatives signifient 0
substr(start, length)	de start obtient length caractères	permet un start negatif

Comparer les strings

Comme nous le savons du chapitre Comparaisons, les strings sont comparées caractère par caractère dans l'ordre alphabétique. Bien que, il y a quelques bizarreries.

Strings

1. Une lettre minuscule est toujours plus grande qu'une majuscule :

```
alert( 'a' > 'Z' ); // true
```

2. Les lettres avec des signes diacritiques sont "hors d'usage" :

```
alert('Österreich'>'Zealand');// true
```

Cela peut conduire à des résultats étranges si nous trions ces noms de pays. Habituellement, les gens s'attendent à trouver Zealand après Österreich dans la liste.

Pour comprendre ce qui se passe, nous devons être conscients que les chaînes de caractères en JavaScript sont encodées en utilisant UTF-16. C'est-à-dire que chaque caractère a un code numérique correspondant.
Il existe des méthodes spéciales qui permettent d'obtenir le caractère pour le code et inversement :

str.codePointAt(pos)
Renvoie un nombre décimal représentant le code du caractère à la position pos :

```
alert(  "Z".codePointAt(0) ); // 90
alert(  "z".codePointAt(0) ); // 122
alert( "z".codePointAt(0).toString(16) ); // 7a (si nous avons besoin d'une valeur hexadécimale)
```

Strings

String.fromCodePoint(code)

Crée un caractère par son code chiffre

```
alert( String.fromCodePoint(90) ); // Z
alert( String.fromCodePoint(0x5a)); // Z
(nous pouvons également utiliser une valeur hexadécimale comme argument)
```

Voyons maintenant les caractères avec les codes 65..220 (l'alphabet latin et un peu plus) en créant une chaîne de caractères de ceux-ci :

```
let str = "";
for (let i = 65; i <= 220; i++) {
  str += String.fromCodePoint(i);
}
alert( str );
// Output:
// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~□□□□□□
// ¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ
```

Strings

Vous voyez ? Les caractères majuscules sont les premiers, puis quelques spéciaux, puis les minuscules, et Ö vers la fin de la sortie.
Maintenant, cela devient évident pourquoi a > Z.
Les caractères sont comparés par leur code numérique. Le plus grand code signifie que le caractère est plus grand. Le code pour a (97) est supérieur au code pour Z (90).
Toutes les lettres minuscules vont après les lettres majuscules car leurs codes sont plus grands.
Certaines lettres comme Ö se distinguent de l'alphabet principal. Ici, le code est supérieur à tout ce qui va de a à z.

Les comparaisons correctes
L'algorithme "approprié" pour effectuer des comparaisons de chaînes est plus complexe qu'il n'y paraît, car les alphabets diffèrent d'une langue à l'autre.
Le navigateur doit donc connaître la langue à comparer.
Heureusement, les navigateurs modernes prennent en charge la norme d'internationalisation ECMA-402.
Elle fournit une méthode spéciale pour comparer des chaînes de caractères dans différentes langues, en respectant leurs règles.

Strings

L'appel str.localeCompare(str2) renvoie un entier indiquant si str est inférieur, égal ou supérieur à str2 selon les règles du langage :

- Renvoie un nombre négatif si str est inférieur à str2
- Renvoie un nombre positif si str est supérieur à str2
- Renvoie 0 s'ils sont équivalents.

Par exemple :

```
alert(  'Österreich'.localeCompare('Zealand') ); // -1
```

Cette méthode a en fait deux arguments supplémentaires spécifiés dans la documentation, ce qui lui permet de spécifier la langue (par défaut, pris dans l'environnement, l'ordre des lettres dépend de la langue) et de définir des règles supplémentaires telles que la sensibilité à la casse ou doit-on traiter "a" et "á" de la même manière, etc.

Arrays

Les objets vous permettent de stocker des collections de valeurs à clé. C'est très bien.
Mais assez souvent, nous trouvons qu'il nous faut une collection ordonnée, où nous avons un 1er, un 2ème, un 3ème élément, etc. Par exemple, nous avons besoin de cela pour stocker une liste de quelque chose : utilisateurs, trucs, éléments HTML, etc.
Il n'est pas pratique d'utiliser un objet ici, car il ne fournit aucune méthode pour gérer l'ordre des éléments. Nous ne pouvons pas insérer une nouvelle propriété "entre" celles existantes. Les objets ne sont tout simplement pas destinés à un tel usage.
Il existe une structure de données spéciale appelée Array (tableau), pour stocker les collections ordonnées.

Déclaration

Il existe deux syntaxes pour créer un tableau vide :

```
let arr = new Array();
let arr = [];
```

La plupart du temps c'est la deuxième syntaxe qui est utilisée. Nous pouvons fournir des éléments initiaux entre parenthèses :

```
let fruits = ["Pomme", "Orange", "Banane"];
```

Arrays

Les éléments de tableau sont numérotés en commençant par zéro.
On peut obtenir un élément par son numéro grace aux crochets :

```
let fruits = ["Pomme", "Orange", "Banane"];
alert( fruits[0] ); // Pomme
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Banane
```

Nous pouvons remplacer un élément :

```
fruits[2] = "Poire"; // maintenant ["Pomme", "Orange", "Poire"]
```

...Ou en ajouter un nouveau au tableau :

```
fruits[3] = "Citron"; // maintenant ["Pomme", "Orange", "Poire","Citron"]
```

Le nombre total d'éléments dans le tableau est sa **length** (longueur) :

```
let fruits = ["Pomme", "Orange", "Banane"];
alert( fruits.length ); // 3
```

Nous pouvons également utiliser un alert pour afficher l'ensemble du tableau :

```
let fruits = ["Pomme", "Orange", "Banane"];
alert( fruits); // "Pomme", "Orange", "Banane"
```

Un tableau peut stocker des éléments de tout type.
Par exemple :

Arrays

Un tableau peut stocker des éléments de tout type. Par exemple :

```
// mélange de valeurs
let arr = [ 'Apple', { name: 'John' }, true,
function() { alert('hello'); } ];

// récupère l'objet à l'index 1 et montre ensuite
son nom
alert( arr[1].name ); // John
// affiche la fonction à l'index 3 et l'exécute la
arr[3](); // hello
```

Les méthodes pop/push, shift/unshift

Une queue (file d'attente) est l'une des utilisations les plus courantes pour les tableaux. En informatique, cela signifie une collection ordonnée d'éléments qui supporte deux opérations :

- **push** ajoute un élément à la fin.
- **shift** enlève un élément depuis le début, en faisant avancer la file d'attente, de sorte que le deuxième élément devienne le premier.

Arrays

Les tableaux prennent en charge les deux opérations.

En pratique, nous en avons besoin très souvent. Par exemple, une file d'attente de messages devant être affichés à l'écran.
Il y a un autre cas d'utilisation pour les tableaux – la structure de données nommée stack.
Il supporte deux opérations :

- **push** ajoute un élément à la fin.
- **pop** enlève un élément de la fin.

Ainsi, de nouveaux éléments sont ajoutés ou enlevés toujours à partir de la "fin".

Un stack (une pile) est généralement illustrée par un jeu de cartes. De nouvelles cartes sont ajoutées ou enlevées par le haut. Pour les stacks, le dernier élément envoyé est reçu en premier, c'est le principe *LIFO* (Last-In-First-Out, dernier entré, premier sorti). Pour les files d'attente, nous avons FIFO (First-In-First-Out, premier entré, premier sorti).
Les tableaux en JavaScript peuvent fonctionner à la fois en queue et en stack. Ils vous permettent d'ajouter ou supprimer des éléments à la fois par le début ou par la fin.

Arrays

Méthodes qui fonctionnent avec la fin du tableau :

pop
Extrait le dernier élément du tableau et le renvoie :

```
let fruits = ["Pomme", "Orange", "Banane"];
alert( fruits.pop() ); // Supprime "Banane"
alert( fruits ); // "Pomme", "Orange"
```

Les deux méthodes fruits.pop() et fruits.at(-1) renvoient le dernier élément du tableau, mais fruits.pop() modifie également le tableau en supprimant l'élément.

push
Ajoute l'élément à la fin du tableau :

```
let fruits = ["Pomme", "Orange", "Banane"];
fruits.push("Poire"); // Ajoute "Poire"
alert( fruits ); // "Pomme", "Orange",  "Poire"
```

Méthodes qui fonctionnent avec le début du tableau :

shift
Extrait le premier élément du tableau et le renvoie :

```
let fruits = ["Pomme", "Orange", "Banane"];
alert( fruits.shift() ); // Supprime "Pomme"
alert( fruits ); // "Orange", "Banane"
```

Arrays

unshift
Extrait le premier élément du tableau et le renvoie :

```
let fruits = ["Orange", "Banane"];
fruits.unshift("Pomme"); // Ajoute "Pomme"
alert( fruits ); // "Pomme", "Orange",  "Banane"
```

Les méthodes **push** et **unshift** peuvent ajouter plusieurs éléments à la fois :

```
let fruits = ["Pomme"];
fruits.push("Orange", "Pêche");
fruits.unshift("Ananas", "Citron");
alert( fruits ); // "Ananas", "Citron","Pomme","Orange", "Pêche"
```

Boucles
L'une des méthodes les plus anciennes pour cycler des éléments de tableau est la boucle for sur les index :

```
let fruits = ["Pomme","Orange", "Banane"];
for (let i = 0; i < fruits.length; i++) {
   alert( fruits[i] );
}
```

Arrays

Mais pour les tableaux, il existe une autre forme de boucle, **for..of** :

```
let fruits = ["Pomme","Orange", "Banane"];
for (let fruit of fruits)) {
   alert( fruit );
}
```

Le **for..of** ne donne pas accès au numéro de l'élément actuel, mais à sa valeur, mais dans la plupart des cas, cela suffit. Et c'est plus court.
Techniquement, comme les tableaux sont des objets, il est également possible d'utiliser **for..in** :

```
let fruits = ["Pomme","Orange", "Banane"];
for (let key in fruits)) {
   alert( fruits[ key ] );
}
```

Arrays

Mais c'est en fait une mauvaise idée. Il y a des problèmes potentiels avec cela :

1. La boucle **for..in** itère sur toutes les propriétés, pas seulement les propriétés numériques.
   Il existe des objets dits "array-like" dans le navigateur et dans d'autres environnements, qui ressemblent à des tableaux. C'est-à-dire qu'ils ont les propriétés length et index, mais ils peuvent également avoir d'autres propriétés et méthodes non numériques, dont nous n'avons généralement pas besoin. La boucle for..in les listera cependant. Donc, si nous devons travailler avec des objets de type tableau, ces propriétés "supplémentaires" peuvent devenir un problème.
2. La boucle **for..in** est optimisée pour les objets génériques, pas pour les tableaux, elle est 10-100 fois plus lente. Bien sûr, c'est encore très rapide. L'accélération peut n'importer que dans les goulots d'étranglement ou sembler hors de propos. Mais il faut quand même être conscient de la différence.

En règle générale, nous ne devrions pas utiliser for..in pour les tableaux.

Un mot à propos de "length"

La propriété length est automatiquement mise à jour lorsque nous modifions le tableau. Pour être précis, il ne s'agit pas du nombre de valeurs du tableau, mais du plus grand index numérique plus un.

Par exemple, un seul élément avec un grand index donne une grande longueur :

Arrays

```
let fruits = [];
fruits[ 123 ]  = "Pomme";
alert( fruits.length ); // 124
```

Notez que nous n'utilisons généralement pas de tableaux de ce type.

Une autre chose intéressante à propos de la propriété **length** est qu'elle est accessible en écriture.
Si nous l'augmentons manuellement, rien d'intéressant ne se produit. Mais si nous le diminuons, le tableau est tronqué. Le processus est irréversible, voici l'exemple :

```
let arr = [1, 2, 3, 4, 5];
arr.length = 2;  // tronque à 2 éléments
alert( arr );    // [1, 2]
arr.length = 5;  // retourne la length d'origine
alert( arr[3] ); // undefined: les valeurs ne reviennent pas
```

Ainsi, le moyen le plus simple pour effacer le tableau est arr.length = 0;.

Arrays

new Array()

Il y a une syntaxe supplémentaire pour créer un tableau :

```
let arr = new Array("A", "B", "etc");
```

Il est rarement utilisé, car les crochets [] sont plus courts. En outre, il comporte une caractéristique délicate.
Si **new Array** est appelé avec un seul argument qui est un nombre, il crée un tableau sans éléments, mais avec la longueur donnée.
Voyons comment on peut se tirer une balle dans le pied :

```
let arr = new Array(2);
alert( arr[0] );     // undefined! pas d'éléments.
alert( arr.length ); // length 2
```

Arrays

Nous connaissons déjà des méthodes qui ajoutent et suppriment des éléments au début ou à la fin :

- arr.push(...items) – ajoute des éléments à la fin,
- arr.pop() – supprime un élément à la fin,
- arr.shift() – supprime un élément au début,
- arr.unshift(...items) – ajouter des éléments au début.

En voici quelques autres.

splice

Comment supprimer un élément du tableau ?

Les tableaux sont des objets, nous pouvons donc utiliser **delete** :

```
let arr = ["Je", "suis", "grand"];
delete arr[1]; // supprime "suis"
alert( arr[1] ); // undefined
// maintenant arr = ["Je",  , "grand"];
alert( arr.length ); // 3
```

L'élément a été supprimé, mais le tableau a toujours 3 éléments, on peut voir que arr.length == 3

C'est normal, car **delete obj.key** supprime une valeur par la clé. C'est tout ce que ça fait. C'est donc parfait pour les objets. Mais pour les tableaux, nous souhaitons généralement que le reste des éléments se déplace et occupe la place libérée. Nous nous attendons à avoir un tableau plus court maintenant.

Arrays

Des méthodes spéciales doivent donc être utilisées.

La méthode **arr.splice** est un couteau suisse pour les tableaux.
Elle peut tout faire : *ajouter, supprimer et remplacer* des éléments.

La syntaxe est la suivante :

```
arr.splice(start[, NbElASupprimer, elem1, ..., elemN])
```

Il a modifié arr à partir de l'index start : supprime les éléments NbElASupprimer puis insère elem1, ..., elemN à leur place. Renvoie le tableau des éléments supprimés.
Cette méthode est facile à comprendre avec des exemples.
Commençons par la suppression :

Par exemple :

```
let arr = ["J", "apprend", "JavaScript"];
arr.splice(1, 1); // À partir de l'index 1 supprime 1 élément
alert( arr ); // ["J", "JavaScript"]
```

Facile, non ? À partir de l'index 1, il a supprimé 1 élément.

Arrays

Dans l'exemple suivant, nous supprimons 3 éléments et les remplaçons par les deux autres :

```
let arr = ["J", "apprend", "JavaScript","depuis", "peu",];
// supprime les 3 premiers éléments et les remplace par d'autre
arr.splice(0, 3, "On", "connait");
alert( arr ) // maintenant ["On", "connait", "right", "now"]
```

Nous pouvons voir ici que splice renvoie le tableau des éléments supprimés :

```
let arr = ["J", "apprend", "JavaScript","depuis", "peu",];
// supprime les 2 premiers éléments
let removed = arr.splice(0, 2);
alert( removed ); // "J", "apprend" <-- tableau des éléments supprimés
```

La méthode splice est également capable d'insérer les éléments sans aucune suppression. Pour cela, nous devons défi
NbElASupprimer sur 0 :

```
let arr = ["J", "apprend", "JavaScript"];
// de l'index 2 on  supprime 0 et ajoute "langage" et "complexe"
arr.splice(2, 0, "langage", "complexe");
alert( arr ); // "J","apprend","langage","complexe","JavaScript"
```

Arrays

slice

La méthode arr.slice est beaucoup plus simple qu'un similaire arr.splice.

La syntaxe est la suivante :

```
arr.slice( [start], [end])
```

Il retourne un nouveau tableau dans lequel il copie tous les éléments index qui commencent de start à end (sans compter end). Les deux start et end peuvent être négatifs, dans ce cas, la position depuis la fin du tableau est supposée.

Cela ressemble à une méthode string str.slice, mais au lieu de sous-chaînes de caractères, cela crée des sous-tableaux.

Par exemple :

```
let arr = ["t", "e", "s", "t"];
alert( arr.slice(1, 3) ); // e,s (copie de 1 à 3, 3 non compris)
alert( arr.slice(-2) ); // s,t (copie de -2 jusqu'à la fin)
```

Nous pouvons aussi l'appeler sans arguments : **arr.slice()** créer une copie de arr. Cela est souvent utilisé pour obtenir une copie pour d'autres transformations qui ne devraient pas affecter le tableau d'origine.

Arrays

Nous pouvons aussi l'appeler sans arguments : arr.slice() créer une copie de arr. Cela est souvent utilisé pour obtenir une copie pour d'autres transformations qui ne devraient pas affecter le tableau d'origine.

concat

La méthode arr.concat crée un nouveau tableau qui inclut les valeurs d'autres tableaux et des éléments supplémentaires.

La syntaxe est la suivante :

```
arr.concat(arg1, arg2...)
```

Il accepte n'importe quel nombre d'arguments – des tableaux ou des valeurs.

Le résultat est un nouveau tableau contenant les éléments arr, puis arg1, arg2, etc.

Si un argument argN est un tableau, alors tous ses éléments sont copiés. Sinon, l'argument lui-même est copié.

Par exemple :

```
let arr = [1, 2];
// créer un tableau à partir de arr et [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4
// créer un tableau à partir de arr et [3,4] et [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6
// créer un tableau à partir de arr et [3,4], puis ajoute les valeurs 5 et 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Arrays

Itérer: forEach

La méthode arr.forEach permet d'exécuter une fonction pour chaque élément du tableau.

La syntaxe :

```
arr.forEach(function(item, index, array) {
  // ... fait quelques chose avec l'élément
});
```

Par exemple, cela montre chaque élément du tableau :

```
// pour chaque élément appel l'alerte
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

Et ce code est plus élaboré sur leurs positions dans le tableau cible :

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`${item} est à l'index ${index} dans ${array}`);
});
```

Arrays

Recherche dans le tableau

Voyons maintenant les méthodes de recherche dans un tableau.

indexOf/lastIndexOf et includes

Les méthodes arr.indexOf, et arr.includes ont la même syntaxe et utilisent essentiellement la même chose que leurs équivalents de chaîne, mais fonctionnent sur des éléments au lieu de caractères :

- **arr.indexOf(item, from)** recherche l'élément item à partir de l'index from, et retourne l'index où il a été trouvé, sinon il retourne -1.
- **arr.includes(item, from)** – recherche l'élément item en commençant par l'index from, retourne true si il est trouvé.

Habituellement, ces méthodes sont utilisées avec un seul argument : l'élément à rechercher. Par défaut, la recherche s'effectue depuis le début.

Par exemple :

```
let arr = [1, 0, false];
alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1
alert( arr.includes(1) ); // true
```

Arrays

Veuillez noter que indexOf utilise l'égalité stricte === pour la comparaison. Donc, si nous cherchons "faux", il trouve exactement "faux" et non le zéro.
Si nous voulons vérifier si item existe dans le tableau et n'avons pas besoin de l'index, alors arr.includes est préféré.
La méthode arr.lastIndexOf est la même que indexOf, mais recherche de droite à gauche.

```
let fruits = ['Pomme', 'Orange', 'Pomme']
alert( fruits.indexOf('Pomme') ); // 0 ( Première pomme )
alert( fruits.lastIndexOf('Pomme') ); // 2 (Dernière 'Pomme)
```

find et findIndex/findLastIndex

Imaginez que nous ayons un tableau d'objets. Comment pouvons-nous trouver un objet avec une condition spécifique ?
Ici la méthode arr.find(fn) se révèle vraiment pratique. La syntaxe est la suivante :

```
let result = arr.find(function(item, index, array) {
 // devrait retourner true si l'élément correspond à ce que nous recherchons
 // pour le scénario de "falsy" (faux), renvoie undefined
});
```

Arrays

La fonction est appelée pour chaque élément du tableau, l'un après l'autre :

- item est l'élément.
- index est sont index.
- array est le tableau lui même.

S'il renvoie true, la recherche est arrêtée, l'item est renvoyé. Si rien n'est trouvé, undefined est renvoyé.

Par exemple, nous avons un tableau d'utilisateurs, chacun avec les champs id et name. Trouvons le premier avec l'id == 1 :

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];
let user = users.find(item => item.id == 1);
alert(user.name); // John
```

Dans la vie réelle, les tableaux d'objets sont une chose courante, la méthode find est donc très utile.

Notez que dans l'exemple, nous fournissons à find la fonction item => item.id == 1 avec un argument. C'est typique, les autres arguments de cette fonction sont rarement utilisés.

Arrays

La méthode arr.findIndex est essentiellement la même, mais elle retourne l'index où l'élément a été trouvé à la place de l'élément lui-même. La valeur de -1 est retournée si rien n'est trouvé.
La méthode arr.findLastIndex est comme findIndex, mais recherche de droite à gauche, similaire à lastIndexOf.
Voici un exemple :

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"},
  {id: 4, name: "John"}
];
// Trouver l'index du premier John
alert(users.findIndex(user => user.name == 'John')); // 0
// Trouver l'index du dernier John
alert(users.findLastIndex(user => user.name == 'John')); // 3
```

filter

La méthode find recherche un seul (le premier) élément qui rend la fonction true.
S'il y en a plusieurs, nous pouvons utiliser arr.filter(fn).
La syntaxe est à peu près identique à celle de find, mais filter renvoie un tableau d'éléments correspondants :

Arrays

```
let results = arr.filter(function(item, index, array) {
  // si true, l'item est poussé vers résultats et l'itération continue
  // retourne un tableau vide si rien n'est trouvé
});
```

Par exemple :

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

// retourne les tableaux des deux premiers users
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

Arrays

Transformer un tableau

Passons aux méthodes qui transforment et réorganisent un tableau.

map

La méthode arr.**map** est l'une des plus utiles et des plus utilisées.
Elle appelle la fonction pour chaque élément du tableau et renvoie le tableau de résultats.
La syntaxe est :

```
let result = arr.map(function(item, index, array) {
  // renvoie la nouvelle valeur au lieu de l'item
});
```

Par exemple, ici nous transformons chaque élément en sa longueur :

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length)
alert(lengths); // 5,7,6
```

sort(fn)

La méthode arr.sort trie le tableau en place, en changeant son ordre d'élément. Elle renvoie également le tableau trié, mais la valeur renvoyée est généralement ignorée, comme arr est lui-même modifié.

Arrays

Par exemple :

```
let arr = [ 1, 2, 15 ];
// la méthode réordonne le contenu de arr
arr.sort();
alert( arr );  // 1, 15, 2
```

Avez-vous remarqué quelque chose d'étrange dans le résultat ?
L'ordre est devenu 1, 15, 2. C'est incorrect. Mais pourquoi ?
Les éléments sont triés en tant que chaînes par défaut.
Littéralement, tous les éléments sont convertis en chaînes de caractères pour comparaisons. Pour les chaînes de caractères, l'ordre lexicographique est appliqué et donc "2" > "15".
Pour utiliser notre propre ordre de tri, nous devons fournir une fonction comme argument de arr.**sort**().
La fonction doit comparer deux valeurs arbitraires et renvoyer le résultat :

```
function compare(a, b) {
  if (a > b) return 1; // if the first value is greater than the second
  if (a == b) return 0; // if values are equal
  if (a < b) return -1; // if the first value is less than the second
}
```

Arrays

Par exemple :

```
let arr = [ 1, 2, 15 ];
// la méthode réordonne le contenu de arr
arr.sort();
alert( arr );  // 1, 15, 2
```

Avez-vous remarqué quelque chose d'étrange dans le résultat ?
L'ordre est devenu 1, 15, 2. C'est incorrect. Mais pourquoi ?
Les éléments sont triés en tant que chaînes par défaut.
Littéralement, tous les éléments sont convertis en chaînes de caractères pour comparaisons. Pour les chaînes de caractères, l'ordre lexicographique est appliqué et donc "2" > "15".
Pour utiliser notre propre ordre de tri, nous devons fournir une fonction comme argument de arr.**sort**().
La fonction doit comparer deux valeurs arbitraires et renvoyer le résultat :

```
function compare(a, b) {
  if (a > b) return 1; // if the first value is greater than the second
  if (a == b) return 0; // if values are equal
  if (a < b) return -1; // if the first value is less than the second
}
```

Arrays

Par exemple, pour trier sous forme de nombres :

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 2, 15 ];
arr.sort(compareNumeric);
alert(arr);  // 1, 2, 15
```

Maintenant, ça fonctionne comme nous l'avons prévu.
Mettons cela de côté et regardons ce qui se passe. L'arr peut être un tableau de n'importe quoi, non ? Il peut contenir des nombres, des chaînes de caractères, des objets ou autre. Nous avons donc un ensemble de quelques items. Pour le trier, nous avons besoin d'une fonction de classement qui sache comment comparer ses éléments. La valeur par défaut est un ordre de chaîne de caractères.
La méthode arr.sort(fn) intégre l'implémentation d'un algorithme générique de tri. Nous n'avons pas besoin de nous préoccuper de son fonctionnement interne (c'est un tri rapide optimisé la plupart du temps). Il va parcourir le tableau, comparer ses éléments à l'aide de la fonction fournie et les réorganiser. Tout ce dont nous avons besoin est de fournir la fn qui effectue la comparaison.

Arrays

Une fonction de comparaison peut renvoyer n'importe quel nombre
En réalité, une fonction de comparaison est requise uniquement pour renvoyer un nombre positif pour dire "plus grand" et un nombre négatif pour dire "plus petit". Cela permet d'écrire des fonctions plus courtes :

```
let arr = [ 15, 1, 2 ];
arr.sort( function(a, b) { return a - b; } );
alert(arr);  // 1, 2, 15
```

Souvenez-vous des fonctions fléchées ? Nous pouvons les utiliser ici pour un tri plus net :

```
let arr = [ 15, 1, 2 ];
arr.sort( (a, b) => a - b );
alert(arr);  // 1, 2, 15
```

Arrays

reverse
La méthode arr.reverse inverse l'ordre des éléments dans l'arr.
Il retourne également le tableau arr après l'inversion.

```
let arr = [ 1, 2, 3, 4, 5 ];
arr.reverse();
alert(arr);  // 5,4,3,2,1
```

split et join
Voici une situation réelle. Nous écrivons une application de messagerie et la personne entre dans la liste des destinataires délimités par des virgules : Jean, Pierre, Marie. Mais pour nous, un tableau de noms serait beaucoup plus confortable qu'une simple chaîne de caractères. Alors, comment l'obtenir ?

```
let names = 'Jean, Pierre, Marie';
let arr = names.split(', ');
for (let name of arr) {
  alert( `Un message à ${name}.` );
  // Un message à ...
}
```

La méthode **str.split(separator)** fait exactement cela. Elle divise la chaîne en un tableau selon le délimiteur separator donné.
La méthode **split** a un deuxième argument numérique facultatif – une limite sur la longueur du tableau. S'il est fourni, les éléments supplémentaires sont ignorés. En pratique, il est rarement utilisé cependant :

```
let names = 'Jean, Pierre, Marie, Isa';
let arr = names.split(', ', 2);
alert( arr ); // Jean, Pierre
```

Arrays

Divisé en lettres

L'appel de **split(s)** avec un **s** vide diviserait la chaîne en un tableau de lettres :

```
let str = "test";
alert( str.split('') ); // t,e,s,t
```

L'appel de **arr.join(separator)** fait l'inverse de **split**. Elle crée une chaîne de caractères avec les éléments de arr joints entre eux par separator. Par exemple :

```
let arr = 'Jean, Pierre, Marie';
let str = arr.join(';'); // joint les éléments en une string en utilisant le caractère ";"
alert( str ); // 'Jean;Pierre;Marie
```

reduce/reduceRight

Lorsque nous devons parcourir un tableau, nous pouvons utiliser **forEach, for** ou **for..of**.
Lorsque nous devons itérer et renvoyer les données pour chaque élément, nous pouvons utiliser map.
Les méthodes **arr.reduce** et **arr.reduceRight** appartiennent également à cette famille, mais sont un peu plus complexes. Ce méthodes sont utilisées pour calculer une valeur unique basée sur un tableau.
La syntaxe est la suivante :

```
let value = arr.reduce(function(accumulator, item, index, array) {
  // ...
}, [initial]);
```

Arrays

La fonction est appliquée à tous les éléments du tableau les uns après les autres et "reporte" son résultat à l'appel suivant.
Les arguments :

- accumulator – est le résultat de l'appel de fonction précédent, égal à initial la première fois (si initial est fourni).
- item – est l'élément actuel du tableau.
- index – est sa position.
- array – est le tableau.

Lorsque la fonction est appliquée, le résultat de l'appel de fonction précédent est transmis au suivant en tant que premier argument.

Ainsi, le premier argument est l'accumulateur qui stocke le résultat combiné de toutes les exécutions précédentes. À la fin, il devient le résultat de la fonction **reduce**.

Arrays

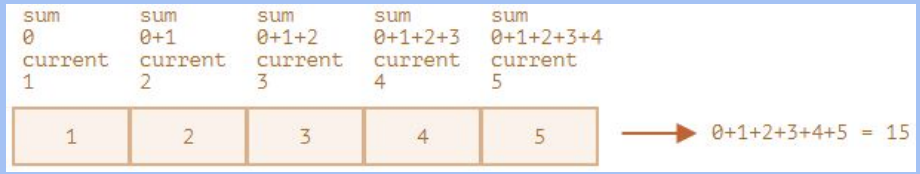
Cela semble compliqué ?
Le moyen le plus simple pour comprendre c'est avec un exemple.
Ici nous obtenons la somme d'un tableau sur une ligne :

```
let arr = [1, 2, 3, 4, 5];
let result = arr.reduce( (sum, current) => sum + current, 0 );
alert( result ); // 15
```

La fonction passée à reduce utilise seulement 2 arguments, c'est généralement suffisant.
Voyons en détails ce qu'il se passe.

1. Lors du premier passage, sum prend la valeur de initial (le dernier argument de reduce), égale à 0, et current correspond au premier élément du tableau, égal à 1. Donc le résultat de la fonction est 1.
2. Lors du deuxième passage, sum = 1, nous y ajoutons le deuxième élément du tableau (2) et sum est retourné.
3. Au troisième passage, sum = 3 et nous y ajoutons un élément supplémentaire, et ainsi de suite...

Le flux de calcul :



Arrays

Ou sous la forme d'un tableau, où chaque ligne représente un appel de fonction sur l'élément de tableau suivant :

	sum	current	result
premier appel	0	1	1
deuxième appel	1	2	3
troisième appel	3	3	6
quatrième appel	6	4	10
cinquième appel	10	5	15

Ici, nous pouvons clairement voir comment le résultat de l'appel précédent devient le premier argument du suivant.
Nous pouvons également omettre la valeur initiale :

```
let arr = [1, 2, 3, 4, 5];
// Suppression de la valeur initiale de reduce (pas de 0)
let result = arr.reduce((sum, current) => sum + current);
alert( result ); // 15
```

Le résultat est le même. En effet, s'il n'y a pas de valeur initiale, alors reduce prend le premier élément du tableau comme valeur initiale et lance l'itération à partir du deuxième élément.
Le tableau de calcul est le même que celui ci-dessus, sans la première ligne.

Map

Jusqu'à présent, nous avons découvert les structures de données complexes suivantes :
Les objets sont utilisés pour stocker des collections de clés.
Les tableaux sont utilisés pour stocker des collections ordonnées.
Mais ce n'est pas suffisant pour la vie réelle. C'est pourquoi Map et Set existent également.

Map

Une Map est une collection d'éléments de données saisis, tout comme un Object. Mais la principale différence est que Map autorise les clés de tout type.

Voici les méthodes et les propriétés d'une Map :

- new Map() – créer la map.
- map.set(key, value) – stocke la valeur par la clé.
- map.get(key) – renvoie la valeur par la clé, undefined si key n'existe pas dans la map.
- map.has(key) – retourne true si la key existe, false sinon.
- map.delete(key) – supprime l'élément (la paire clé/valeur) par la clé.
- map.clear() – supprime tout de la map.
- map.size – renvoie le nombre d'éléments actuel.

Arrays

Par exemple :

```
let map = new Map();
map.set('1', 'str1');   // une clé de type chaîne de caractère
map.set(1, 'num1');     // une clé de type numérique
map.set(true, 'bool1'); // une clé de type booléenne
// souvenez-vous, dans un `Object`, les clés sont converties en chaîne de caractères
// alors que `Map` conserve le type d'origine de la clé,
// c'est pourquoi les deux appels suivants retournent des valeurs  :
alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'
alert( map.size ); // 3
```

Au travers de cet exemple nous pouvons voir, qu'à la différence des **Objects**, les clés ne sont pas converties en chaîne de caractère. Il est donc possible d'utiliser n'importe quel type.
map[key] n'est pas la bonne façon d'utiliser un Map
Bien que **map[key]** fonctionne également, par exemple nous pouvons définir **map[key]** = 2, cela traite map comme un objet JavaScript simple, ce qui implique toutes les limitations correspondantes (uniquement des clés chaîne de caractères/symbol etc.).
Nous devons donc utiliser les méthodes de map : **set**, **get** et ainsi de suite.

Arrays

Itération dans Map

Il existe 3 façons de parcourir les éléments d'une map :

- map.keys() – renvoie un itérable pour les clés,
- map.values() – renvoie un itérable pour les valeurs,
- map.entries() – renvoie un itérable pour les entrées [key, value], il est utilisé par défaut dans for..of.

L'ordre d'insertion est conservé
Contrairement aux Object, Map conserve l'ordre d'insertion des valeurs.

Object.entries: Créer une Map à partir d'un objet

Lorsqu'une Map est créée, nous pouvons passer un tableau (ou un autre itérable) contenant des paires clé/valeur pour l'initialisation, comme ceci :

```
let map = new Map();
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);
// on parcourt les clés (les légumes)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // cucumber, tomatoes, onion
}
// on parcourt les valeurs (les montants)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}
// on parcourt les entries (couple [clé, valeur])
for (let entry of recipeMap) { // équivalent à :
recipeMap.entries()
  alert(entry); // cucumber,500 (etc.)
}
```

Arrays

```
// tableau de paires [clé, valeur]
let map = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);
alert( map.get('1') ); // str1
```

Si nous avons un objet simple et que nous souhaitons en créer une Map, nous pouvons utiliser la méthode intégrée Object.entries(obj) qui renvoie un tableau de paires clé/valeur pour un objet exactement dans ce format.

Nous pouvons donc créer une Map à partir d'un objet de la manière suivante :

```
let obj = { name: "John", age: 30 };
let map = new
Map(Object.entries(obj));
alert( map.get('name') ); // John
```

Arrays

Ici, Object.entries renvoie le tableau de paires clé/valeur : [["name", "John"], ["age", 30]]. C'est ce dont a besoin la Map.

Object.fromEntries: Objet à partir d'une Map

Nous venons de voir comment créer une Map à partir d'un objet simple avec Object.entries(obj).

Il existe une méthode Object.fromEntries qui fait l'inverse : étant donné un tableau de paires [clé, valeur], elle crée un objet à partir de ces paires :

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);
// maintenant, prices = { banana: 1,
orange: 2, meat: 4 }
alert(prices.orange); // 2
```

Arrays

Nous pouvons utiliser Object.fromEntries pour obtenir un objet simple à partir d'une Map.
Par exemple, nous stockons les données dans une Map, mais nous devons les transmettre à un code tiers qui attend un objet simple.
Voici comment procéder :

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);
let obj = Object.fromEntries(map.entries()); // créer un objet
simple (*)
// terminé!
// obj = { banana: 1, orange: 2, meat: 4 }
alert(obj.orange); // 2
```

Un appel à map.entries() renvoie un itérable de paires clé/valeur, exactement dans le bon format pour Object.fromEntries.
Nous pourrions également raccourcir la ligne (*) :

```
let obj = Object.fromEntries(map); // .entries() omis
```

Arrays

Set

Un Set est une collection de types spéciaux – "ensemble de valeurs" (sans clés), où chaque valeur ne peut apparaître qu'une seule fois.
Ses principales méthodes sont :

- new Set([iterable]) – crée le set et si un objet iterable est fourni (généralement un tableau), en copie les valeurs dans le set.
- set.add(value) – ajoute une valeur, renvoie le set lui-même.
- set.delete(value) – supprime la valeur, renvoie true si value existait au moment de l'appel, sinon false.
- set.has(value) – renvoie true si la valeur existe dans le set sinon false.
- set.clear() – supprime tout du set.
- set.size – c'est le nombre d'éléments.

Ce qu'il faut surtout savoir c'est que lorsque l'on appelle plusieurs fois set.add(value) avec la même valeur, la méthode ne fait rien. C'est pourquoi chaque valeur est unique dans un Set.

Par exemple, nous souhaitons nous souvenir de tous nos visiteurs. Mais chaque visiteurs doit être unique. Set est exactement ce qu'il nous faut :

Arrays

```
let set = new Set();
let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };
// visites, certains utilisateurs viennent plusieurs fois
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);
// set conserve une fois chaque visiteurs
alert( set.size ); // 3
for (let user of set) {
  alert(user.name); // John (puis Pete et Mary)
}
```

L'alternative à Set aurait pu être un tableau d'utilisateurs en vérifiant avant chaque insertion que l'élément n'existe pas en utilisant arr.find. Cependant les performances auraient été moins bonnes car cette méthode parcours chaque élément du tableau. Set est beaucoup plus efficace car il est optimisé en interne pour vérifier l'unicité des valeurs.

Arrays

Parcourir un Set

Nous pouvons parcourir les éléments d'un Set avec for..of ou en utilisant forEach :

```
let set = new Set(["oranges", "apples", "bananas"]);
for (let value of set) alert(value);
// même chose en utilisant forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

A noter que la fonction de callback utilisée par forEach prend 3 arguments en paramètres : une value, puis la même valeur valueAgain, et enfin le set lui-même.
C'est pour la compatibilité avec Map où le callback forEach passé possède trois arguments. Ça a l'air un peu étrange, c'est sûr. Mais cela peut aider à remplacer facilement Map par Set dans certains cas, et vice versa.
Les méthodes pour parcourir les éléments d'une Map peuvent être utilisées :

- set.keys() – renvoie un objet itérable pour les valeurs,
- set.values() – identique à set.keys(), pour compatibilité avec Map,
- set.entries() – renvoie un objet itérable pour les entrées [value, value], existe pour la compatibilité avec Map.