Apprendre à programmer en JS

Killian Reine

Table des matières

1 Les variables

1.1 Généralités sur les variables

Définition

Une **variable** permet de stocker une donnée.
Une **donnée** est une information, que l'on range dans une variable. Elle possède un **nom** qui permet de l'identifier et un **type**, qui indique la nature de l'élément stocké (lettre, nombre, ...).

Déclarer une variable

En JavaScript, il est possible de déclarer des variables de plusieurs manières différentes.

- Variable statique

```
const nomVariable = valeur ;
```

Une variable déclarée avec le mot clé `const` est non-mutable, c'est à dire qu'elle ne peut pas changer de valeur.

- Variable dynamique

```
let nomVariable = valeur ;
```

Une variable déclarée avec le mot clé `let` est mutable, on peut donc la changer.

Si vous souhaitez afficher des résultats dans la console, il suffirat d'utiliser :

```
console.log(nomVariable) ;
```

1.2 Types courrant de variable

En JavaScript, il existe plusieurs types de variable. Les plus courrants sont :

- Les nombres `Number`
- Les caractères et les chaines de caractères `String`
- Les booléens, vrai ou faux `Boolean`

Exemple 1.1

```
let prenom = "killian";
let age = 18;
const sexe = "masculin";
let majeur = true;
```

Complément sur le type `String`

Remarque

Les chaîne de caractères peuvent être données entre simple guillemet, double guillements, ou même entre backtick que l'on utilise en MarkDown et sur discord.

Ajouter une apostrophe

```
let chaine = 'Une chaine d\'aujourd'hui';
```

Ajouter une variable

```
let chaine = 'prenom : ${variable}';
```

Une chaine sur plusieurs lignes

```
let paragraphe = `je suis
sur
plusieurs lignes`;
```

1.3 Oppérations sur les variables

1.3.1 Oppérations arithmétique

Exemple 1.2

```
let a = 5;
let b = 6;
let sum = a + b; //Addition
console.log(sum);
```

```
11
```

```
let diff = a - b; //Différence
console.log(diff);
```

```
-1
```

```
let prod = a * b; //Produit
console.log(prod);
```

```
30
```

```
let div = a/b; //Division, quotient
console.log(div);
```

```
0.83333333333
```

```
let mod = a % b; //Modulo
console.log(mod);
```

```
5
```

Remarque

Les oppérations d'assignations peuvent aussi être utilisées : `+=`, `-=`, `*=`, `/=` `%=`

1.3.2 Incrémentation et décrémentation

Exemple 1.3

```
let a = 0;
let b = 3;
a++;
b--;
console.log(a, b);
```

```
1 2
```

1.3.3 Opération de comparaison

Les opérations de comparaison valides.

- Inférieur strict <, Inférieur ou égal <=
- Supérieur strict >, Supérieur ou égal >=
- Égalité ==, égalité **stricte** ===
- Différence !=, différence **stricte** !==

Zoom

Comparaison d'égalité

Exemple 1.4

```
let a = 12;
let b = "12";
let c = 12;
console.log(a==b, a===b, a===c);
```

```
true false true
```

Comparaison de différence

Exemple 1.5

```
let a = 12;
let b = "12";
let c = 12;
console.log(a=!b, a==!b, a==!c);
```

```
false true false
```

La comparaison stricte tient compte du type des variables mises en jeu

Remarque

La syntaxe //... pour insérer des commentaires.

1.3.4 Opérations logique

Exemple 1.6

ET logique

```
let a = true;
let b = true;
console.log(a && b); //renvoie true
```

OU logique

```
console.log(a || b); //renvoie true
```

NON logique

```
console.log(!a); //renvoie false
```

Si besoin, petit rappel sur les tables de vérités du ET, OU et NON logique.

1.4 Concaténation des chaînes de caractères

Exemple 1.7

```
let chaine = "Bonjour";
let chaine2 = "le monde";
let nombre = 47;
console.log(chaine+=chaine2);
```

```
Bonjourle monde
```

```
console.log(chaine+nombre);
```

```
Bonjour47
```

et oui, en JavaScript on peut ajouter des chaines e de caractères et des nombres ensembles.

1.5 Conversion de variable

Conversion en entier

Pour convertir une chaine de caractère en nombre :

```
let nombre = Number(chaine);
```

Exemple 1.8

```
let c1 = "2";
let nb = 4;
console.log(Number(c1);
```

```
2
```

```
console.log(nb+c1, nb+Number(c1));
```

```
"42", 6
```

1.6 Types plus complexes

Les types suivants seront étudiés en détail plus tard dans le cours.

1.6.1 Les objets

Définition

Un **objet** permet de stocker diverses informations de différents types dans un même endroit.

Stucture générale d'un objet

```
const nomObjet = {
   elem1 : val1,
   elem2 : val2,
   ⋮
   elemn : valn
};
```

Exemple 1.9
On vas créer un objet élève et on vas stocker ses informations personnelles ainsi que ses notes.

```
const eleve = {
   nom : "Reine",
   prenom : "Killian",
   sexe : "M",
   dateNaissance : "16/09/2005",
   age : 18,
   notes : [12, 15, 7, 20]
};
console.log(eleve.nom, eleve['nom']);
```

"Reine", "Reine"

```
console.log(eleve.notes[3], eleve['notes'][1]);
```

20 15

Remarque

Les objets peuvent contenir eux aussi des objets.

1.6.2 Les tableaux

Définition

Les **tableaux** permettent de stocker une liste d'information. Cette liste peut contenir n'importe quel autre type de variable (un tableau peut même contenir un autre tableau).

Structure générale d'un tableau

```
monTableau = [elt1, elt2, ..., eltn];
```

Méthodes associées aux tableaux

- accéder à l'élément *n*

```
monTableau[n];
```

- ajouter un élément à la fin du tableau

```
monTableau.push(nvValeur);
```

- Supprimer le dernier élément

```
monTableau.pop( );
```

- afficher la taille du tableau

```
monTableau.length;
```

Remarque

Une liste peut elle même contenir une liste, c'est ce qu'on appelle tableau à *n*-dimentions.

1.7 Les types spéciaux

- `undefined` est le type qui apparaîtra lorsque vous essayer d'accéder à une valeur inexistante.
  Par exemple quand vous souhaitez accéder à un élément d'un tableau mais que vous êtes sorti de ce dernier.
- `null` absence de valeur.
- `NaN` Not a Number.

2 Copie par valeur, copie par référence

2.1 Copie par référence

Lorsque l'on copie une variable par **référence**, dans notre cas, la variable `copieListeInitiale` pointe vers **la même liste** que `listeInitiale` c'est la raison pour laquelle lorsque `listeInitiale` est modifiée alors `copieListeInitiale` l'est aussi.

Exemple 2.1

```
let listeInitiale = [1, 2, 3];
let copieListeInitiale = listeInitiale;
//Affichage des listes avant modification
console.log(listeInitiale, copieListeInitiale);
```

```
[1, 2, 3] [1, 2, 3]
```

```
//Modification de la liste initiale
listeInitiale.push(4);
//Affichage des listes après modification
console.log(listeInitiale, copieListeInitiale);
```

```
[1, 2, 3, 4] [1, 2, 3, 4]
```

2.2 Copie par valeur

Lors de la création de la variable `copieListeInitiale`, on utilise la fonction `list( )` qui vas en **fait créer une nouvelle liste** contenant la même chose que `liste Initiale`.

Exemple 2.2

```
let listeInitiale = [1, 2, 3];
let copieListeInitiale = Array.from(listeInitiale);
//Affichage des listes avant modification
console.log(listeInitiale, copieListeInitiale);
```

```
[1, 2, 3] [1, 2, 3]
```

```
//Modification de la liste initiale
listeInitiale.push(4);
//Affichage des listes après modification
console.log(listeInitiale, copieListeInitiale);
```

```
[1, 2, 3, 4] [1, 2, 3]
```

3 Les conditions

Structure de la condition `if`

```
if ( condition <1> ) {
   instructions si <1> vrai;
} else if ( condition <2> ) {
   instructions si <2> vrai;
} else {
   instruction si <1> et <2> sont faux;
}
```

Remarque

- On a vu les différents opérateurs de comparaison
- **Attention**, lorsque l'on compare des objets, même si ces derniers ont les mêmes propriétés, il ne sont pas considérés

Exemple 3.1

```
{ a : 1 } == { a : 1 }; //false
{ } == { }; //false
== [ ]; //false
NaN == NaN; //false
```

Saisie utilisateur

Pour permettre à l'utilisateur de saisir des informations dans une boite de dialogue :

```
let saisie = prompt("Tapez quelque-chose : ");
```

3.1 Le switch / case

Structure de la switch/case

```
switch (valeurATester) {
   case proposition<1> :
      instruction si valeurATester == proposition<1>;
   case proposition<2> :
      instruction si valeurATester == proposition<2>;
      break
   case proposition<n> :
      instruction si valeurATester == proposition<n>;
   default :
      instruction à effectuer par défaut;
}
```

Au lieu de faire n conditions `if` et `else if`, vous pouvez utiliser cette structure qui permet aussi de gérer les exceptions (= erreurs).

3.2 Le ternaire

Définition

Le **ternaire** est un opérateur conditionnel qui permet de réduire une condition du type `if else` en une seule ligne.

Structure du ternaire

```
let ternaire = (condition ? <valeur si vrai> : <valeur si faux>);
```

Exemple 3.2

```
//Déterminer si je suis majeur ou mineur
const age = 18;
const statut = (age >=18) ? "majeur" : "mineur";
console.log(statut);
```

```
"majeur"
```

4 Portée des variables

Définition

- Une **variable globale** peut être utilisé partout dans notre code, elle est souvent déclarée au début du code.
- Une **variable locale** peut être utilisé dans un bloc de code bien précis, les paramètres d'une fonction sont des variables locales utilisables uniquement à l'intérieur de celle-ci.

Remarque

On peut avoir deux variables qui comporte le même nom **mais** une portée différente.

Exemple 4.1

```
let variable = 2; //Variable GLOBALE
if ( true ) {
   let variable = 5; //Variable LOCALE
   console.log("variable dans le if : ", variable);
}
console.log("variable hors du if : ", variable);
```

```
variable dans le if : 5
variable hors du if : 2
```

5 Les boucles

5.1 La boucle `while`

La boucle `while` permet d'exécuter un même code en boucle **tant qu'une condition est vraie**.
Attention, à bien faire attention à la condition d'arrêt (qui devra forcément devenir faux) de votre boucle `while` au risque de faire une boucle infinie...

Structure de la boucle `while`

```
while ( condition ) {
   instructions tant que condition est vraie;
}
```

On peut aussi mettre la condition en fin de boucle avec `do...while`

```
do {
   Instructions tant que condition est vraie;
} while ( condition )
```

forcer la sortie d'une boucle

```
break;
```

5.2 La boucle `for`

La boucle `for` permet d'exécuter un code un certain nombre de fois en précisant manuellement l'intervalle pour lequel on souhaite faire la boucle.

Structure de la boucle `for`

```
for ( let variable = valInitiale ; condition ; incrementer variable ) {
  // Instructions
}
```

5.2.1 Boucle `for...in`

La boucle `for...in` permet d'itérer sur les éléments énumérables. Elle permettra de récupérer les clefs d'un tableau ou les propriétés d'un objet

Structure du `for...in`

```
for ( const var in objet ) {
  // instructions
}
```

5.2.2 Boucle `for...of`

La boucle `for...of` permet de boucler sur un objet itérable en renvoyant les valeurs à chaque itération.

Structure du `for...of`

```
for ( const var of objet ) {
  // instruction à itérer sur chaque éléments de objet
}
```

6 Les fonctions

Définition

Les **fonctions** permettent de stocker en mémoire une certaine logique que l'on pourra utiliser à plusieurs reprises dans la suite de notre code. Elles prennent en général des paramètres et retournent un résultat particulier.

Strucure d'une fonction

```
function nomFonction ( parametres ) {
  return;
}
//Pour appeler la fonction
nomFonction( parametres );
```

Remarque

Les fonctions en JavaScript sont un type de variable particulier, il est donc aussi possible de ne pas leur donner de nom, mais **de les stocker dans une variable** de manière classique.

```
const nomFonction = function ( parametres ) {
   return;
};
```

6.1 Portées des variables

Une fonction déclarée dans une variable aura la même portée que les variables (limité au bloc courant). Par contre une fonction déclarée directement avec le mot clef function aura une portée plus globale.
Aussi, **l'hoisting** fera qu'une fonction peut être appelée avant d'être déclarée.

Définition

Le terme "**hosting**" en JavaScript fait référence à la manière dont les variables sont stockées et accessibles dans différents environnements, tels que les fonctions ou les blocs de code. En termes simples, c'est l'endroit où une variable est déclarée et où elle peut être utilisée dans votre code.
En JavaScript, il existe deux types de hosting :

- hosting de variable
- hosting de fonction

6.1.1 Hosting de variable

Lorsque vous déclarez une variable avec `var`, `let`, ou `const`, elle est "hébergée" au début de la portée dans laquelle elle est déclarée, mais elle reste "indéfinie" jusqu'à ce que l'exécution du code atteigne la ligne où elle est déclarée.

Exemple 6.1

```
console.log(x); // Renvoie undefined
var x = 10;
console.log(x); // Renvoie 10
```

En réalité, le code ci-dessus est interprété comme suit par le moteur JavaScript :

```
var x;
console.log(x); // Renvoie undefined
x = 10;
console.log(x); // Renvoie 10
```

6.1.2 Hosting de fonction

Les fonctions sont également "hébergées" au début de leur portée, ce qui signifie que vous pouvez appeler une fonction avant sa déclaration dans le code.

Exemple 6.2

```
sayHello(); // Renvoie "Bonjour!"
function sayHello() {
   console.log("Bonjour!");
};
```

Revient à coder :

```
function sayHello() {
   console.log("Bonjour!");
}
sayHello(); // Renvoie "Bonjour!"
```

6.2 L'opérateur `this`

Exemple 6.3

```
function nomFonction ( ) {
   console.log(this);
}
nomFonction.call(4);
```

```
4
```

Dans notre cas, `this` prendra la valeur du paramètre de la fonction.

Exemple 6.4

```
const variable = {
   prop : 42,
   maFonction : function( ){
      return this.prop;
   },
};
console.log(a.maFonction( ));
```

```
42
```

6.3 Les fonctions fléchées

Les fonctions fléchées sont une syntaxe alternative (plus courte) pour les fonctions.
Ces fonctions ont comme particularité de ne pas posséder de valeur `this`.

Exemple 6.5

```
const maVariable = (parametre) => {
   return parametre+4;
}
maVariable( 2 );
```

```
6
```

Remarque

si il n'y a qu'une instruction de retour, on pourra simplifier l'appel en retirant les accolades. On pourra aussi retirer les parenthèses si il n'y a qu'un paramètre.

Exemple 6.6

```
const double = (n) => {
   return 2 * n;
}
// On peut simplifier en retournant directement en retirant les accolades
const double = (n) => 2 * n;
// Et on peut retirer les parenthèses
const double = n => 2 * n;
```

6.4 Fonctions usuelles

Et oui, certaines méthodes / fonctions sont à connaîtres car très souvent utilisées. Certaines ressemblent à ceux déjà vues en python par exemple.

Trancher une chaine de caractère

```
maChaine.split(separateur);
```

Mettre le texte en majuscule

```
maChaine.toUpperCase( );
```

Inverser les éléments dans une liste

```
maListe.reverse( );
```

Mettre le texte en minuscule

```
maChaine.toLowerCase( );
```

Refusionner les éléments d'une liste

```
maChaine.join(fusionneur);
```

Remplacer tout les même mots

```
maChaine.replaceAll(mot, remplace);
```

Exemple 6.7

```
const chainePrincipale = "Bienvenue à vous";
const chaineReverse = chainePrincipale.reverse( );
console.log("chaineReverse :", chaineReverse);
const chaineListe = chainePrincipale.split(" ");
console.log("chaineListe :", chaineListe);
const chaineMaj = chainePrincipale.toUpperCase( );
console.log("chaineMaj :", chaineMaj);
const chaineMin = chainePrincipale.toLowerCase( );
console.log("chaineMin :", chaineMin);
const chaineU = chainePrincipale.replaceAll("u", "OUU");
console.log("chaineU :", chaineU);
```

```
chaine reverse : erreur
chaineListe : ["B", "i", "e", "n", "v", "e", "n", "u", "e", " ", "à", " ", "v", "o",
"u", "s"]
chaineMaj : "BIENVENUE A VOUS"
chaineMin : "bienvenue à vous"
chaineU : "BienvenOUUe à voOUUs"
```

Remarque

Pour inverser une chaine de caractère, il ne suffit pas de taper `chaine.reverse()`. Ceci, vous renverra une erreur (voir exemple du dessus).
Pour inverser une liste :

- Convertir la chaine principale en liste avec `split( )`
- Inverser la liste avec `reverse( )`
- Ré-assembler la liste pour obtenir la chaine résultante inversée avec `join( )`

7 Les classes

7.1 Introduction aux prototypes

Pour rappel, JavaScript est un langage de programmation orienté objet, mais il se distingue des autres langages car il utilise un système de **prototype** au lieu de classes.

Définition

En JavaScript, chaque objet possède un **lien vers un autre objet** appelé **prototype**. D'ailleur, un prototype peut lui-même avoir un prototype et ainsi de suite, ce qui forme donc une chaine de prototypes. Ce mécanisme est utilisé pour l'héritage.

Accéder au prototype d'un objet

Soit `objet` un objet créé et initialisé au préalable, afin d'accéder à son prototype, on utilisera l'instruction suivante :

```
Object.getPrototypeOf(objet)
```

Nous pouvons bien sûr récupérer le prototype d'une fonction, d'une classe avec cette instruction.

Les codes déclarés (créés) avec `class` et `function` renvoient une fonction `[[Prototype]]`. Avec les prototypes, chaque fonction peut devenir une instance de constructeur en utilisant `new`.

Remarque

Rappelons qu'une **instance** désigne en fait une copie de la fonction/class qui possède ces propres valeurs et qui permet d'accéder aux méthodes associées à la class principale.

7.2 Création d'objets et de prototypes

7.2.1 Fonction de constructeur

Les **fonctions constructeurs** permettent de créer des instances qui possèdent les mêmes propriétés.

Exercice 7.1

```
function Personne (prenom, age) {
   this.prenom = prenom;
   this.age = age;
}
Personne.prototype.bonjour = function( ) {
   return 'Bonjour, je m'appelle ${this.prenom} et j'ai ${this.age} ans.';
}

// Création d'une instance de classe
const alice = new Personne("Alice", 50);
console.log(alice.bonjour( ));
```

```
Bonjour, je m'appelle Alice et j'ai 50 ans.
```

7.2.2 Constructeur de class

Exemple 7.2, *renvoi la même chose que l'exemple 7.1*

```
class Personne {
   constructor(prenom, age) {
      this.prenom = prenom;
      this.age = age;
   }
   bonjour {
      return 'Bonjour, je m'appelle ${this.prenom} et j'ai ${this.age} ans.';
   }
}
const alice = new Personne("Alice", 50);
console.log(alice.bonjour( ));
```

Remarque

Grâce aux exemples 7.1 et 7.2, nous venons de voir comment assigner des méthodes à une fonction de constructeur et à une class.

Fonction constructeur avec assignation d'une méthode

```
 // Fonction de constructeur
function fonctionConstructeur(parametres){
   // Code permettant l'initialisation du constructeur
}

// Assignation d'une méthode
fonctionConstructeur.prototype.nomMethode = function(parametres) {
   // Code de la méthode associée à la fonction de constructeur
}
```

Class avec un constructeur et une méthode

```
 class nomClass {
   constructor(parametres) {
      // Code permettant l'initialisation du constructeur
   }
}
   nomMethode = function(parametres) {
      // Code de la méthode
   }
}
```

Remarque

Le constructeur permet en fait d'initialiser les propriétés, attributs d'un objet. Cela garantit que l'objet en question est exécuté au départ de manière cohérente et sans erreurs.
Un peu plus tard, nous observerons le rôle des constructeurs dans l'héritage.

7.2.3 Les getters et les setters

Définition

- Les **getters** sont des mini-méthodes qui permettent d'**accéder** au contenu des propriétés d'un objet
- Les **setters** eux, permettent de **modifier** la valeur des propriétés d'un objet.

Cela garantit alors la protection des données internes d'un objet en contrôlant comment on peut y accéder et les modifier.

Structure d'un getter et d'un setter

```
class Personne {
   // Constructeur de ma class
   constructor(prenom, age) {
      this._prenom = prenom;
      this._age = age;
   }
   get variable( ){
      return this._variable
   }
   set variable(nouv){
      this._prenom = nouv; return this._variable
   }
}
```

Remarque

Les getters et setters ne peuvent pas avoir le même nom qu'une propriété.
Comme vous l'avez sûrement remarqué, juste après les `this` de notre constructeur nous utilisons la forme `_propriété`. L'underscore "_" est **une convention** en JavaScript qui permet d'indiquer que les propriétés / variables de la class ne sont utilisable qu'en interne.
Pour faire court qu'elles ne devraient pas êtres accessible ou modifiable directement hors de la class.

7.3 Propriété publique, propriété privée

On vient donc d'évoquer l'underscore (_) qui **permet seulement de spécifier** que l'utilisation des propriétés de la classe ne peut s'en faire qu'en interne.
Malheureusement, comme écrit cette convention ne fais que spécifier (= indiquer) que l'utilisation doit se faire ainsi, au final ça ne change rien sur l'utilisation des propriétés puisqu'elles sont quand même modifiable hors de la class.
Cependant, ES6 introduit les champs privé grâce au préfixe #.

ES6

ES6, plus connu sous le nom de ECMAScript 2015 est la sixième édition du langage ECMAScript qui constitue une mise à jour majeure de JavaScript.
Publiée en juin 2015 par ECMA International, cette version introduit de nombreuses fonctionnalités et amélioration du langage permettant ainsi le développement de logiciel plus complexe et maintenable.

Exemple 7.3

```
class Personne {
   constructor(prenom, age){
      this.#prenom = prenom;
      this.#age = age;
   }
   get prenom( ){
      return this.#prenom;
   }
   set prenom(nouveauPrenom){
      this.prenom = nouveauPrenom;
   }
}
const alice = new Personne("Alice", 50);
console.log(alice.#prenom);
```

```
erreur
// on ne peut plus accéder directement aux propriétés, il faut passer par les getters et setters
```

```
 // Cette fois, on accède à la propriété prénom, en utilisant un getters
console.log(alice.prenom);
```

```
Alice
```

```
 alice.prenom = bob;
console.log(alice.prenom);
```

```
erreur
// Vous essayez de modifier la valeur de prenom sans passer par un setters
```

7.4 Introduction à l'héritage de class

L'héritage permet dans les langages orienté objet la réutilisation du code, en créant de nouvelles class basées sur celles déjà existantes en réutilisant les propriétés, méthode de cette dernière.
La class sur laquelle la nouvelle s'appuie sera appelée **class parente**.

7.4.1 Exemple avec les fonctions de constructeur

Exemple 7.4
Nous allons reprendre notre fonction de constructeur (exemple 7.1) `Personnage` et nous allons créer une nouvelle sous-fonction que nous nomerons `superHeros` avec un nouvel attribut qui sera le pouvoir.

```
function superHeros(prenom, age, pouvoir) {
   Personnage.call(this, prenom, age);
   this.pouvoir = pouvoir;
}
```

7.4.2 Exemple avec les class

Les class, introduites pas ES6 utilisent le mot clé `super` au lieu de `call` pour accéder aux fonctions parents. Pour renvoyer à la class parents, on utilise `extends`.

Exemple 7.5
Même principe que l'exemple précédant, sauf que là, on prend la class Personnage.

```
class superHeros extends Personne {
   constructor (prenom, age, pouvoir) {
      super(prenom, age);
      this.pouvoir = pouvoir;
   }
}
```

8 Gestion des erreurs

Jusqu'à lors, nous n'avons pas pris compté des erreurs qu'il peut exister dans notre code. Puisque lorsque l'on code des fonctions, elles ont souvent des paramètres d'un type voulu. Mais vous avez sûrement remarqué que aucun système ne vérifie la validité des paramètres donnés.

Exemple 8.1
Un professeur d'informatique vient de coder une fonction qui calcul la moyenne de ses élèves.

```
function moyenne(notes) {
   nbNote = notes.length;
   sommeNote = 0;
   for(i = 0; i<notes.length; i++) {
      sommeNote += notes[i];
   }
   return 'Ta moyenne est de ${sommeNote/nbNote} /20';
}
console.log(moyenne[15, 20, 5, 0]);
console.log(moyenne(25, 12, 14));
```

Ta moyenne est de 10/20
Ta moyenne est de **NaN**/20

Comme vous le voyez, la fonction codée necessite d'entrer un tableau de note en paramètre. Malgré tout, lorsque l'on entre des nombres à l'aveugle, **la fonction s'exécute quand même**, malgré que le résultat obtenu ne soit pas celui attendu.
L'objectif ici est donc que la fonction retourne une erreur lorsque l'entrée `notes` n'est pas un tableau.
Nous allons donc rajouter un test.

Exemple 8.1

```
function moyenne(notes) {
   if (!Array.isArray(notes)) {
      throw new Error('Les notes doivent être contenues dans un tableau');
   }
   nbNote = notes.length;
   sommeNote = 0;
   for(i = 0; i<notes.length; i++) {
      sommeNote += notes[i];
   }
   return 'Ta moyenne est de ${sommeNote/nbNote} /20';
}
console.log(moyenne[15, 20, 5, 0]);
console.log(moyenne(25, 12, 14));
```

```
Ta moyenne est de 10/20
ERROR
```

Jusqu'ici, nous avons réussi à faire *planter* notre programme si les notes ne sont pas un tableau. Mais moi je veux que loe code affiche le message d'erreur souhaité.
On utilise alors `try ...catch ....`
On reprend notre code, en ajoutant les modifications.

```
try {
   console.log(moyenne[15, 20, 5, 0]);
   console.log(moyenne(25, 12, 14));
}
catch (e) {
   console.error(e.message); // Afficher le message d'erreur
}
```

```
Ta moyenne est de 10/20
Les notes doivent êtres contenues dans un tableau.
```

8.1 Vérification des types

Opérateur `typeof`

Il est possible d'afficher le type d'une valeur (variable) en utilisant la forme suivante :

```
console.log(typeof variable);
```

Exemple 8.2
La fonction suivante renvoie `true` si le paramètre donné est de type `number`, `false` sinon.

```
function verifNombre(nb){
   type = typeof nombre;
   return type=="number";
}
```

8.2 Vérification spécifique des objets

Objet de type `array`

Dans l'exemple global, nous avons vu comment vérifié si une variable est un objet de type `array` (= tableau).

```
Array.isArray(variable)
```

Objet de type `Date`

L'instruction suivante permet de vérifier si `variable` est bien une instance d'objet `Date`.

```
variable instanceof Date
```

Remarque

La forme `instanceof` fonctionne aussi pour vérifier si une variable est une instance de :

- `RegExp`, une expression régulière
- `Error`, une erreur

Les informations relatives à ces nouveaux objets encore inconnus seront explicitées dans la partie suivante.

Vérification `NaN`, Not-a-number

```
Number.isNaN(variable)
```

9 Objets et types avancés

9.1 Objet `Date`

Définition

En JavaScript, `Date` est un objet utilisé pour **manipuler les dates et les heures**. Ce dernier est associé à de nombreuses méthodes permettant de formater, créer, comparer et calculer des dates et des heures.

Méthode associées à l'objet `Date`

- Récuperer la date actuelle

```
let variable = new Date( );
```

- Récupérer une date spécifique

```
let dateSpecifique = new Date('annee-mois-jour');
```

- Utiliser les arguments séparés

```
let dateSpecifique = new Date(annee, mois, jour);
```

- Timestamp, milliseconde depuis le **1er janvier 1970**, UTC

```
let dateSpecifique = new Date(nbMillisecondes);
```

Remarque

Il est aussi possible de récupérer l'année uniquement, ou autre grâce aux méthodes suivantes :

- `date.getFullYear( );`
- `date.getMonth( );`
- `date.getDate( );`
- `date.getDay( );`
- `date.getHours( );`
- `date.getMinutes( );`
- `date.getSeconds( );`
- `date.getMilliseconds( );`

Attention, lorsque vous voulez vous même définir la date, vous remplacer `get` par `set` mais prennez garde, les mois sont numérotés comme suit 0 pour janvier et 11 pour décembre !

9.2 Les expressions régulières

Définition

Les **expressions régulières** (regex) sont des motifs, utilisés pour trouver ou manipuler du texte dans des chaînes de caractères.
Elles permettent ainsi de rechercher des mots, caractères, motifs spécifiques, remplacer, extraires, ...

9.2.1 Création d'une expression régulière

Instructions

- Avec une expression

```
let regex = /motif/;
```

- En créant un objet

```
let regex = new RegExp('motif');
```

9.2.2 Recherche de motif dans une chaine de caractère

Exemple 9.1

```
let regex = /abc/;
console.log(regex.test("abcdef"));
console.log(regex.test("defg"));
```

```
true
false
```

9.2.3 Renvoyer toute les correspondances

Exemple 9.2

```
let regex = /(a)(bc)/;
let chaineTest = "cdefabc";
let result = regex.exec(chaineTest);
console.log(result);
```

```
['abc', 'a', 'bc', index : 4, input : 'cdefabc', groups : undefined]
```

```
let regex2 = /(abc)d/;
let result2 = regex2.exec(chaineTest);
console.log(result2);
```

```
null
```

9.2.4 Renvoyer l'indice de correspondance

Exemple 9.3

```
let regex = /(a)(bc)/;
let chaineTest = "cdefabc";
let result = chaineTest.search(regex);
console.log(result);
```

```
4
```

```
let regex2 = /(abc)d/;
let result2 = chaineTest.search(regex2);
console.log(result2);
```

```
-1
```

9.2.5 Remplacement de motif

Exemple 9.4

```
let str = 'abcdef';
let nouvStr = str.replace(/abc/, 'xyz');
console.log(nouvStr);
```

```
xyzdef
```

Remarque

Il existe aussi les **classes de caractères**

- `/[abc]/`
  a, b ou c
- `/[^abc]/`
  tout les caractères sauf a, b ou c

Les expressions régulières peuvent être utilisées lors de la validation d'un email, pour vérifier si le format est bien respecté.

Validité d'un email

```
let emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
console.log(emailRegex.test('example@example.com'));
console.log(emailRegex.test('invalid-email'));
```

```
true
false
```

9.3 La valeur `NaN`

Définition

`NaN` (Not a Number) est une valeur spéciale qui indique qu'une opération a échouée ou alors qu'une valeur n'est pas un nombre valide.

Remarque

Imaginons que votre code convertisse une chaine de caractère en nombre grâce à la fonction number :

```
let nb = Number("du texte");
```

La valeur retournée sera donc `NaN` puisque convertir une chaine de caractère n'est pas possible, l'opération a donc échouée d'où le `NaN`.
Opération valide

```
let nb = Number("1560")
```

10 Méthodes associées aux objets de type string

La méthode `charAt( )`

Lorsque l'on souhaite récupérer un caractère à un indice spécifique dans une chaine, on utilise la méthode `charAt( )` :

```
chaineCaract.charAt(indice)
```

Remarque

Imaginons la chaine `"chat"`, le mot est composé de 4 lettres (indice 0 à 3). Si je sors dépasse l'intervalle des indices, la méthode me renverra `""`, la chaîne vide.

Exemple 10.1

```
let chaine = "étudiant";
console.log(chaine.charAt(0);
console.log(chaine.charAt(8);
```

```
"é"
""
```

Remarque

ECMAScript 5, a introduit une nouvelle manière de récupérer le caractère à l'indice `i` :

```
let lettre = "chaine"[i];
```

Il est aussi possible de **comparer** les caractères.

Exemple 10.2

```
let a="o";
let b="h";
console.log(a<b);
console.log(a>b);
```

```
false
true
```

Égalité de deux chaînes de caractères

Intuitivement, pour vérifier si une chaine `ch1` est égale à une autre chaine `ch2`, vous allez utiliser la forme :

```
ch1 == ch2
```

Malheureusement, si vous avez `ch1="chat"` et `ch2="Chat"`, le programme vous renverra `false` car l'égalité avec == prend en compte la casse du texte.
Voici comment contrer ce problème :

```
function egalite(ch1, ch2){
   return ch1.toUpperCase() === ch2.toUpperCase();
}
```

Type string primitif :

- immuable, c'est à dire qu'elles ne peuvent pas être modifiées après leur création
- elle sont plus légères en terme de ressource puisqu'elles sont stockées dans la pile mémoire directement
- Leur syntaxe est simplifiée entre " ou '.

Exemple 10.3

```
let primi = "Bonjour";
```

Objet string :

- mutable, peuvent être modifiés
- possèdent des méthodes associé
- les objets `string` sont construit à l'aide d'un constructeur.

Exemple 10.4

```
let objetString = new String("Bonjour");
```

Exemple 10.5
Additionner avec des chaines de caractères.

```
let primiString = "2 + 2";
let objetString = new String("2 + 2");
console.log(eval(primiString));
console.log(eval(objetString));
```

```
4
"2 + 2"
```

La fonction intégrée `eval`

La fonction intégrée `eval( )` permet de prendre en argument une chaine de caractère et de l'évaluer comme du code JavaScript.

```
code(expression)
```

Remarque

Un objet string peut tout de même être converti en équivalent de type primitif :

```
objetString.valueOff( );
```

11 Les timers

11.1 setTimeout

Définition

La méthode `setTimeout` est utilisée pour exécuter une fonction (un bloc de code) après un certain temps (exprimé en millisecondes).

Instruction générale de `setTimeout`

```
setTimeout(fonction, delai, ...args)
```

- `fonction`, la fonction à exécuter après le délai.
- `delai`, le délai en **millisecondes** avant exécution de la fonction
- `...args` (facultatifs), des arguments supplémentaires à passer à la fonction

Exemple 11.1.1

```
function direBonjour( ){
   console.log("Bonjour");
}
setTimeout(bonjour, 1000);
```

Exemple 11.1.2

```
setTimeout(direBonjour( ){
   console.log("Bonjour");
}, 1000);
```

Exemple 11.2

```
function direBonjour(prenom){
   console.log('Bonjour ${prenom}');
}
setTimeout(bonjour, 1000, "Alice");
```

Remarque

Il est tout à fait possible d'annuléer un `setTimeout` avec :

```
clearTimeout(timer);
```

11.2 setInterval

Définition

Le méthode `setInterval` est utilisée pour exécuter une fonction ou un bloc de code de manière répétée, à intervalle régulier donné en millisecondes.
`setInterval` continuera d'exécuter le morceau de code tant qu'on ne lui dit pas d'arrêter.

Instruction générale de la méthode `setInterval`

```
setInterval(fonction, intervalle, ...args)
```

- `fonction`, le bloc de code a exécuter
- `intervalle`, au bout de combien de temps le code se répète (millisecondes)
- `...args` (facultatifs), arguments passés à la fonction

Exemple 11.3
```
setInterval(function( ) {
   console.log("Un message à afficher toutes les 3 secondes");
}, 3000);
```

Remarque

On peut aussi annuler / arrêter un `setInterval` grâce à l'instruction :

```
clearInterval(intervalle);
```

12 Code asynchrone

Dans la vie courante, on dit que deux actions sont **synchrone** lorsqu'elles se déroulent simultanément (en même temps) ou de manière comme son nom l'indique synchroniser. Au contraire, deux actions sont **asynchrone** lorsqu'elles ne se déroulent pas en même temps.

Définition

En informatique,

- deux opérations sont dites **synchrone** lorsque la seconde opération attend que la première est terminée pour s'exécuter. Ducoup, le début de l'opération $n+1$ dépend de la complétitude de l'opération n.
  On dit alors que **les opérations sont dépendantes les unes des autres**.
- deux opérations sont donc dites **asynchrone** lorsqu'elles sont **indépendantes** les unes des autres, c'est à dire que l'opération $n+1$ n'a pas besoin de l'opération n pour démarer.

Pour le moment, restons sur le principe de dépendance pour comprendre les définitions de synchrone et d'asynchrone.

Synchrone et asynchrone

Prenons un exemple assez imagé, nous avons été manger dans deux restaurants il y a peu et l'organisation de ces deux restaurants nous ont interpelées.

- Situation asynchrone :
  Les clients sont assis à plusieurs tables, si ils le souhaitent, ils peuvent passer commande en même temps et être servis dès que leur plat sera prêt. En informatique on parle d'opérations asynchrone. Deux prises de commandes sont indépendantes et l'une n'affecte pas l'autre.
- Situation synchrone :
  Maintenant, on imagine que le restaurant ne possède qu'une unique employé qui est à la fois cuisinier et serveur, dans ce cas, il ne peut prendre qu'une commande à la fois et pourra en prendre une autre lorsqu'il aura servit la table courante. Par contre, ici les opérations sont dites synchrones.

Remarque

Pourtant, par défaut, JavaScript est un langage synchrone ce qui signifie que les opérations vont être exécutées les unes à la suite des autres. Chaque opérations doit donc attendre que la précédante ait terminée, l'opération précédante est appelée **opération bloquante**. De plus JavaScript ne peut exécuter qu'une ligne à la fois (le code s'exécute sur un thread, en gros un fil / processus unique).
Cela peut poser des problèmes, par exemple, si une fonction prend trop de temps à s'exécuter, cette dernière vas alors bloquer tout le reste des instructions suivante, le programme va donc sembler « arrêté » du point de vue de l'utilisateur.

En JavaScript, les opérations asynchrones sont placées dans une file d'attente qui vont s'exécuter après que la têche principale (`main thread`) ait terminée ses opérations. Ce qui ne bloque pas le reste du code.

Objectif

L'idée principale du principe d'asynchrone est que le reste du code puisse continuer à s'exécuter pendant qu'une opération, plus longue, qui demande une valeur, ... est en cours. D'un côté navigateur web, cela permet un affichage plus rapide des pages et une meilleure expérience.

Définition

Une **fonction de rappel** (ou callback) est une fonction qui va pouvoir être **rappelée** à un moment et / ou si les conditions sont réunies.
Les fonctions de rappel étaient utilisé par le premier outil en JavaScript qui a introduit la notion de code asynchrone.

Exemple 12.1
Précédemment, nous avons rencontré la méthode `setTimeout( )` qui elle est asynchrone. Ce qui signifie que la suite du code n'a pas besoin d'attendre que le `setTimeout` soit terminé pour s'exécuter.

```
setTimeout(alert, 5000, "Message après 5 secondes"):
alert("Suite du script");
```

```
Suite du script
Message après 5 secondes
```

Remarque

Les fonctions de rappel sont une alternative au code synchrone mais elles possèdent des défauts :

- On ne peut pas savoir quand notre fonction asynchrone aura fini de s'exécuter
- On ne peut donc pas connaître l'ordre d'exécution des fonctions

Cela pose surtout problème si l'exécution d'une fonction asynchrone dépend elle même d'une fonction asynchrone.

On peut se passer de la remarque lorsque l'on a qu'une seule opération asynchrone ou si le peu d'opérations asynchrone qui sont présente sont totalement indépendante.

13 Les promesses, `Promise`

Pendant longtemps, les fonctions de rappel étaient la seule solution pour pouvoir exécuter du code asynchrone. Cependant, en 2015, introduction d'un outil permettant la création et la gestion de code asynchrone, **les promesses** avec l'objet constructeur `Promise`. Les promise sont encore aujourd'hui la solution la plus puissante permettant d'utiliser l'asynchrone dans nos script.

Définition

Une `Promise` est donc un objet qui permet de représenter l'état d'une opération asynchrone. Une `Promise`, comme dans la vie réelle peut être :

- ▶ Opération en cours
- ▶ Opération terminée avec succès
- ▶ Opération échouée

Alors, au lieu d'appeler de nombreuses fonctions de rappel, nous allons créer ou utiliser des fonctions qui vont renvoyer des promesses et nous allons introduire des fonctions de rappel dans les `Promise`. Ce qui permettra alors de connaître l'ordre d'exécution des opérations asynchrone.

Structure d'une `Promise`

```
const promesse = new Promise((resolve, reject) => {
   // Tâches asynchrone à réaliser
   // Appel de resolve( ) si la promesse est résolue (= tenue)
   // Appel de reject( ) si la promesse a échouée
}
```

Remarque

Une `Promise` permet donc de de représenter et manipuler un résultat, un événement futur ce qui nous permet donc de définir ce que va faire le programme à l'avance après une opération asynchrone terminée, qu'elle soit résolue ou échouée.

Dans la majeure partie des cas, nous n'aurons pas besoin de créer nos propre `Promise` en utilisant le contructeur mais plutôt de manipuler les promesse déjà créées. Les promesses vont être utilisé le plus souvent par des API JavaScript réalisant des opérations asynchrones.

Exemple 13.1

Lorsque vous appelez vos amis sur l'appli Discord, l'on vous demande quel caméra utiliser (si il y en a une, et laquelle) et pareil pour le micro.
Sans code asynchrone, la fenêtre du navigateur / appli vas rester bloquée pour l'utilisateur tant que ce dernier n'ai pas cclairement autorisé quel caméra/micro utiliser.

Après la création d'une promesse, cette dernière vas alors posséder deux propriétés :

- `state`, qui représente l'état de la promesse
  En attente (pending), fulfilled (résolue) ou rejected (échouée).
- `result` qui vas pouvoir contenir la valeur de notre choix

Remarque

L'état final d'une promesse **ne peut pas être changé**.

Définition

La méthode `then( )` est utilisée pour obtenir et exploiter le résultat d'une `Promise`.
Autrement dit, on utilise la méthode `then` pour définir ce qu'il se passera si la promesse et résolue ou si elle a échouée.

Exemple 13.2

```
let promesse = new Promise((resolve, rejecte) => {
   let reussite = true;
   if (reussite) {
      resolve("La promesse a été résolue");
   } else {
      reject("La promesse a échouée");
   }
});
promesse
   .then((message) => {
      console.log(message);
   })
   .catch((erreur) => {
      console.log(erreur);
   });
```

```
La promesse a été résolue
```

Définition

La méthode `catch( )` quant-à elle, est utilisée lorsqu'une promesse est rompue.

13.1 Chainage des promesses

Chainer des méthodes c'est les exécuter les unes après les autres. Cette méthode va s'avérer utile pour exécuter des opérations asynchrones les unes à la suite des autres dans un ordre précis.
Cette méthode est possible car la méthode `then( )` renvoie une promesse. On vas donc pouvoir utiliser une méthode `then` sur le résultat renvoyé par la première méthode `then`.

Exemple 13.3
Dans notre exemple on souhaite réaliser deux opérations asynchrones :

- Récupérer des données sur un serveur
- Traiter les données récupérées

```
function recupererDonnees( ){
   return new Promise((resolve, reject) => {
   setTimeout(( ) => {
         const donnees = { id: 1, nom: "Alice" };
         console.log("Données récupérées", donnees);
         resolve(donnees);
      }, 2000);
   });
}
function traiterDonnees(donnees) {
   return new Promise((resolve, reject) => {
      setTimeout(( ) => {
         donnees.traitement = "Données traitées";
         console.log("Données après traitement", donnees);
         resolve(donnees);
      }, 1000);
   });
}
// Chainage des promesses
recupererDonnees( )
   .then((donnees => {
      // Appel de la seconde fonction asynchrone
      return traiterDonnees(donnees);
   })
   .then((donneesTraitees) => {
      // Traitement final après que les données soient traitées
      console.log(Traitement final :", donneesTraitees);
   })
   .catch((erreur) => {
      console.log("Une erreur est survenue !", erreur);
   });
```

Démarage...
2 secondes s'écoulent les données sont récupérées
Données récupérées : id : 1, nom : 'Alice'
1 seconde s'écoule, le traitement est effectuée
Données après traitement : id : 1, nom : 'Alice', traitement : 'Données traitées avec succès'
Traitement final : id : 1, nom : 'Alice', traitement : 'Données traitées avec succès'

Remarque

Le texte en gris dans le résultat du programme ne s'affiche pas, il n'est là que pour nous permettre de mieux comprendre la simulation.

13.2 Composition de promesses

Définition

Composer des fonctions signifie combiner plusieurs fonctions pour en donner une nouvelle.

De la même manière, nous pouvons composer des promesses. Heureusement des méthodes associées au constructeur `Promise( )` existent.

Remarque

Nous, on a vu que l'on pouvait utiliser `resolve( )` et `reject( )` qui nous permettent de créer manuellement des promesses déjà résolues ou échouées et qui vont être utiles pour démarer une chaine manuellement.

La méthode `all( )`

La méthode `all` prend en argument un tableau de promesses et retourne une nouvelle promesse. Cette dernière sera résolue si l'ensemble des promesses contenues dans le tableau sont résolues.
Ce qui signifique que si au moins une des promesse échoue, la nouvelle promesse renvoyée sera échouée.

Structure générale de la méthode `all( )`

```
Promise.all([promesse1, promesse2, ...])
   .then(([val1, val2, ...]) => {
      // Code a exécutée si toute les promesses sont résolues
   })
   .catch((erreur) => {
      // Code a exécutée si une promesse échoue
   });
```

14 La syntaxe `async ... await`

Définition

La déclaration `async function` et le mot clé `await` sont des « **sucres syntaxiques** », autrement dit, ils n'ajoutent pas de nouvelle fonctionnalité à JavaScript, mais ils permettent d'utiliser des promesses avec un code plus intuitif et qui ressemble d'avantage à la syntaxe du langage, à laquelle nous somme habitués.
Ils sont apparus en 2017 et aujourd'hui utilisé par les API modernes.

Utiliser le mot clé `async` devant une fonction va alors faire en sorte que votre fonction retourne une promesse.
Si votre fonction retourne explicitement une valeur, alors cette dernière sera envelopper dans une promesse.

Exemple 14.1

```
async function bonjour( ){
   return "bonjour";
}
console.log(bonjour( ));
```

```
Promise : { bonjour }
```

Remarque

Le mot clé `await` est valide uniquement dans les fonctions asynchrones définies avec `async`.

Le mot clé `await` permet d'intérompre l'exécution d'une fonction asynchrone tant qu'une promesse n'est pas résolue. Ensuite, cette dernière pourra continuer à s'exécuter et renverra sa valeure finale d'exécution.

Exemple 14.2

```
async function test( ) {
   const promesse = new Promise((resolve, reject) => {
      setTimeout(( ) => resolve("Ok"), 2000)
   });
   let resultat = wait promesse;
   console.log(resultat);
}
```

Remarque

`await promise` retourne le résultat d'une promesse qui est résolue (échec ou résolue).

La syntaxe `try ... catch` permet de retourné l'erreur causée par un `await`.

Structure de la syntaxe `try ... catch`

```
try {
   // Code a tester
   // Alerte a retournée si erreur
} catch(erreur) {
   // Affichage de l'erreur
}
```

Remarque

`all( )` est aussi utilisable avec la syntaxe `async/wait`.

15 La méthode `fetch( )`

La méthode `fetch( )`

La méthode `fetch( )` permet de **faire des appels HTTP** afin de récupérer des ressources sur le réseau. Cette dernière utilise le même système que les promesses vues précédemment.

Structure de la méthode `fetch( )`

```
fetch("url", option)
   .then(reponse => {
      // Traitement de la réponse reçue
   });
   .catch(erreur => {
      // Gestion des erreurs
   });
```

La promesse est résolue et retourne un objet `Response`. Cet objet contient des informations détaillées sur la réponse de la requête HTTP, y compris le statut de la réponse, les en-têtes, et les données renvoyées par le serveur.

Remarque

Quelques propriétés associées à l'objet `Response`.

- `ok` est un booléen qui vaut `true` si le statut HTTP est compris entre 200 et 299, ce qui indique que la requête a été traitée avec succès
- `status` renvoi le statut HTTP de la réponse (200 pour OK, 404 pour Not Found)
- `statusText` indique le message associé au statut HTTP (par exemple "OK" pour 200)
- `header`, `url`, `type`, ...

`text( )` et `json( )`

- `json( )` renvoi la réponse au format JSON
- `text( )` renvoi le texte de la réponse

16 Les modules

Pour vos projets, créer l'intégralité du code sur un seul et même fichier n'est pas pratique. C'est la raison pour laquelle les modules sont indispensables, ils vont vous permettre de diviser votre code en différents fichiers, ce pour vous aider à bien l'organiser.
Il est possible d'exporter des variables pour les rendres accessibles via un autre fichier.

Exemple 16.1

```
export const var = "Alicia";
export function maFonction ( ) {
   console.log("Bonjour tout le monde");
}
export class maClass ( ) {
   constructor(param) {
      this.param = param;
   }
}
```

Remarque

Pour un fichier/module il ne peut n'y avoir qu'une **exportation par défaut**.
On utilisera alors `export default...`

Exemple 16.2

Ici, on souhaite importer dans le fichier `import.js` rapidement les informations de l'exemple 16.1 contenues dans le fichier `exemple.js`.

```
import{ var, maFonction, maClass } from "./exemple.js";
```

Pour importer tout le contenu d'un fichier

```
import * from "./exemple.js";
```