JavaScript Avancé

Références

Typage dynamique

En C, C++, Java, le typage est statique. Il est défini à la déclaration de la variable qui ne peut plus changer de type.

En JavaScript, le typage est dynamique, c'est à dire que le type de la variable est défini au moment de l'affectation. Le type peut donc varier dans le temps.

C++ permet d'indiquer le type de la donnée qui sera contenue dans la variable. JavaScript indique le type de la donnée actuellement contenu dans la variable.

Les types

Types primitifs :

- null
- undefined
- number
- bigint
- boolean
- string
- symbol

Tout le reste est objet (Object).

Le type d'une variable peut être obtenu avec **typeof** qui retournera une chaine de caractère représentant le nom du type :

```
const a = 12;
const typeName = typeof a; // "number"
```

Valeurs et références

Les types primitifs sont passés par valeur. Les types complexes sont passés par référence.

Exemple avec des types primitifs :

```
let a = 5 ;
let b = a ;
a = 10 ;
console.log(`A = ${a}, B = {b}`);    // A = 10, B = 5
```

```
let str1 = "Hi" ;
let str2 = str1 ;
str1 = "Hello" ;
console.log(`Str1 = ${str1}, Str2 = {str2}`); // Str1 = Hello, Str2 = Hi
```

Exemple avec un type complexe :

```
const array1 = [1, 2, 3] ;
const array2 = array1;
array1[1] = 0;
console.log(array2);    // [1, 0, 3]
```

Remarque sur le tableau qui est déclaré **const** mais que l'on peut malgré tout modifier.

Tableaux

Comment dupliquer un tableau ?

- Avec une boucle for

```
const array1 = [1, 2, 3];
const array2 = [];

for(const item of array1)
    array2.push(item);

array1[1] = 0 ;
console.log(array2) ; // [1, 2, 3];
```

- Avec l'opérateur de décomposition (Spread syntaxe)

```
const array1 = [1, 2, 3];
const array2 = [...array1];

array1[1] = 0 ;
console.log(array2) ; // [1, 2, 3];
```

Attention toutefois, si le tableau array1 contient des éléments complexes, il contiendra en réalité des références vers ceux-ci. Références qui seront partagés par array2.

Objets JS

Comparaison avec structures C qui définissent un type utilisable pour décrire la structure d'une donnée complexe.

L'objet javascript est une enveloppe qui regroupe des données (comme une structure en C) à un instant T. La structure peut évoluer dans le temps et ne constitue pas un modèle utilisable pour d'autre variables.

Exemple :

```
const student1 = {
    num: "AEI00023",
    firstname: "Franck",
    lastname: "Dupond"
```

```
}

console.log(student1.num); // AEI00023
```

La structure d'un objet JavaScript peut évoluer dans le temps :

```
student1.mail = "franck.dupond@u-bourgogne.fr";

console.log(student1)

/**
    {
        num: "AEI00023",
        firstname: "Franck",
        lastname: "Dupond",
        mail: "franck.dupond@u-bourgogne.fr"
    }
**/
```

Attention, ce qui peut être une force de JavaScript est aussi sa plus grande faiblesse. Car une faute de frappe peut conduire à de longues heures de débogage.

Fonctions

Fonctions modifiables

En JavaScript, les fonctions sont considérées comme des objets stockés dans une variable du même nom :

```
function fonction1()
{
    console.log("Fonction 1");
}

fonction1 = function() {
    console.log("Fonction 2");
}

window.addEventListener("load", fonction1);

/**
Fonction 2
**/
```

Leur contenu peut donc être modifier au cours du temps. Ce qui peut poser des problèmes, par exemple avec une nouvelle faute de frappe ou avec l'import d'une bibliothèque qui propose une fonction du même nom. La dernière version de la fonction chargée sera celle qui supplantera toutes les autres.

Pour éviter cela, il est possible d'utiliser une autre syntaxe pour la déclaration des fonctions :

```
const fonction1 = function ()
{
    console.log("Fonction 1");
}
fonction1 = function() {
    console.log("Fonction 2");
}

/**
Uncaught TypeError : Assignment to constant variable.
**/
```

Attributs

Si les fonctions sont considérées comme des objets, alors il devrait être possible de leur affecter des attributs, comme pour un objet JavaScript.

```
const increment = function()
{
    console.log(increment.value++);
}

increment.value = 1;

setInterval(increment, 1000);

/**
1
2
3
...
**/
```

Mais on pourrait également ajouter des sous fonctions :

```
...

increment.double = function()
{
    console.log(increment.value * 2);
}

increment.double();

/**
2
**/
```

Classes

Les objets JavaScript de base permettent de regrouper des données mais pas de créer un modèle de structure que l'on pourrait utiliser pour créer plusieurs objets avec la même structure.

Les fonctions quant à elles peuvent exécuter du code et sont également considérées comme des objets, avec des attributs et des méthodes.

Constructeur autodéclarant

JavaScript implémente partiellement le paradigme objet avec les constructeurs autodéclarants :

```
const Student = function()
{
    this.num = "";
    this.firstname = "";
    this.lastname = "";
}

const student1 = new Student();
```

Il est ainsi possible de créer des instances de Student qui auront toutes la même structure et des données indépendantes les unes des autres.

Comme toute fonction, un constructeur autodéclarant peut prendre de paramètres qui serviront à initialiser les attributs de l'instance :

```
const Student = function(num, firstname, lastname)
{
    this.num = num;
    this.firstname = firstname;
    this.lastname = lastname;
}

const student1 = new Student("AEI00023", "Franck", "Dupond");
console.log(student1);

/**
{ num: "AEI00023", firstname: "Franck", lastname: "Dupond" }
**/
```

Il est également possible d'ajouter des méthodes :

```
const Student = function(num, firstname, lastname)
{
    this.num = num;
    this.firstname = firstname;
    this.lastname = lastname;
    this.getFullName = function()
    {
        return `${this.firstname} ${this.lastname}`;
```

```
    }
}

const student1 = new Student("AEI00023", "Franck", "Dupond");
console.log(student1.getFullName());

/**
Franck Dupond
**/
```

Créé de cette façon, chaque instance possède des attributs qui lui sont propres. Et ce qui vaut pour les attributs vaut aussi pour les méthodes qui sont déclarées de la même façon :

```
const student1 = new Student("AEI00023", "Franck", "Dupond");
const student2 = new Student("AEI00024", "Mélodie", "Durant");

console.log(student1.getFullName === student2.getFullName);

/**
false
**/
```

En mémoire, chaque instance possède une version qui lui est propre de ses attributs mais également des ses méthodes. Ce qui veut dire que le code de toutes les fonctions est dupliqué en mémoire ! Ceci peut avoir un impact significatif sur les performances de l'application.

Prototype

Le prototype permet de résoudre ce problème en fournissant une partie commune à toutes les instances d'une même classe :

```
const Student = function(num, firstname, lastname)
{
    this.num = num;
    this.firstname = firstname;
    this.lastname = lastname;
}

Student.prototype.getFullName = function()
{
    return `${this.firstname} ${this.lastname}`;
}


const student1 = new Student("AEI00023", "Franck", "Dupond");
const student2 = new Student("AEI00024", "Mélodie", "Durant");

console.log(student1.getFullName === student2.getFullName);

/**
```

```
true
**/
```

Toutes les instances font référence à la même base de code pour getFullName.

Que se passe-t-il si on place une variable dans le prototype ?

Classe

Syntaxe ES6

La syntaxe du constructeur autodéclarant et du prototype ne sont pas toujours aisée à prendre en main lorsque l'on vient d'autres langages tels que le C++ ou le Java.

La norme EcmaScript 6 a introduit une nouvelle syntaxe plus proche de ce que l'on connait dans d'autres langages :

```
class Student
{
    constructor(num, firstname, lastname)
    {
        this.num = num;
        this.firstname = firstname;
        this.lastname = lastname;
    }

    getFullName()
    {
        return `${this.firstname} ${this.lastname}`;
    }
}

const student1 = new Student("AEI00023", "Franck", "Dupond");

console.log(student1.getFullName());
```

Attention, il ne s'agit là que d'un sucre syntaxique qui facilite la lisibilité du code pour les développeurs. Dans les coulisses, c'est toujours un constructeur autodéclarant couplé à un prototype qui est créé.

Héritage

La syntaxe ES6 introduit également le mot clé extends qui facilite grandement la gestion de l'héritage en JavaScript. L'héritage est réalisable à avec la syntaxe prototype mais nécessite une certaine gymnastique de l'esprit.

```
class User
{
    constructor(firstname, lastname)
    {
```

```
        this.firstname = firstname;
        this.lastname = lastname;
    }

    getFullName()
    {
        return `${this.firstname} ${this.lastname}`;
    }
}

class Student extends User
{
    constructor(num, firstname, lastname)
    {
        super(firstname, lastname);

        this.num = num;
    }
}

const student1 = new Student("AEI00023", "Franck", "Dupond");

console.log(student1.getFullName());
```

Le fonction super doit être appelée dans le constructeur de la classe fille afin d'appeler le constructeur de la classe mère.

Accessibilité

Par défaut, tout les attributs et méthodes sont publiques.

Une nouvelle notation JavaScript permet de définir des éléments privés :

```
class User
{
    #firstname;
    #lastname;

    constructor(firstname, lastname)
    {
        this.#firstname = firstname;
        this.#lastname = lastname;
    }

    getFullName()
    {
        return `${this.#firstname} ${this.#lastname}`;
    }
}
```

A présent, les attributs firstname et lastname sont privés et ne sont accessibles que depuis la classe User.

La notion de protected n'existe pas en JavaScript.

Propriétés

Enfin, la syntaxe ES6 permet de créer des propriétés, c'est à dire des accesseurs (get et set) qui s'utilisent comme des variables :

```
class User
{
    #firstname;
    get firstname() {return this.#firstname}
    set firstname(value) {thi.#fistname = value}

    #lastname;

    constructor(firstname, lastname)
    {
        this.#firstname = firstname;
        this.#lastname = lastname;
    }
}

const user1 = new User("Mélodie", "Durant");

user1.firstname = "Jeanne";

console.log(user1.firstname);

/**
Jeanne
**/
```

Notations avancées

Décomposition

La décomposition permet d'extraire des données d'un tableau ou d'un objet :

```
const array = [ 1, 2, 3, 4 ];

const [ premier, second ] = array;

console.log(premier);
console.log(second);

/**
1
2
**/
```

Comment obtenir le reste du tableau ?

Optional chaining ?.

Le chainage optionnel permet d'interrompre l'évaluation de l'expression si la valeur de ce qui se trouve à gauche de l'opérateur ?. est null ou undefined.

Par exemple, lorsque l'on récupère un élément d'une page HTML avec querySelector, la fonction retourne null si le sélecteur CSS fournit ne correspond à aucun élément de la page. Il est donc nécessaire de faire un test avant de manipuler l'élément récupéré :

```
const htmlItem = document.querySelector("h1");

if(htmlItem)
    htmlItem.classList.add("main-title");
```

Le chainage optionnel permet d'éviter les tests à la chaine et de simplifier le code :

```
document.querySelector("h1")?.classList.add("main-title");
```

Si un titre de niveau H1 a été trouvé, la classe CSS main-title est ajoutée, sinon rien ne se passe.

Attention, le chainage optionnel est valide pour des actions mais pas pour des affectations :

```
document.querySelector("h1")?.innerHTML = "Hello World !"; // Erreur
```

Le code ci-dessus produira une erreur de syntaxe car JavaScript ne saura pas comment gérer l'affectation si l'expression à gauche de ?. vaut null ou undefined.

Nullish coalescing ??

Pour savoir si une variable est différente de null ou undefined, il est tentant d'écrire le code ci-dessous :

```
class Picture
{
    #width;
    #height;

    constructor(width, height)
    {
        if(width)
            this.#width = width;
        else
            this.#width = 1920;

        //Même chose avec l'opérateur ternaire
        this.#height = height ? height : 1080;
    }
}
```

Le problème est qu'en JavaScript, lorsque l'on utilise pas les opérateur === et !== le type de la donnée n'est pas pris en compte, uniquement sa valeur.

Ainsi false, null, undefined, 0, NaN et "" sont considérées comme équivalent à false. Et toute autre valeur est équivalent à true.

Dans l'exemple précédent, si aucune dimension n'est donnée au constructeur de la classe Picture, une résolution par défaut de 1920 x 1080 sera appliquée. Mais si l'on souhaite créer une image de 0 x 0 pixels, l'image finale aura également une résolution de 1920 x 1080 pixels.

L'opérateur ?? permet de résoudre ce problème :

```
class Picture
{
    #width;
    #height;

    constructor(width, height)
    {
        this.#width = width ?? 1920;
        this.#height = height ?? 1080;
    }
}
```

L'opérateur ?? teste la valeur de l'expression à sa gauche. Si cette dernière est différente de null et de undefined alors elle est retournée, sinon, c'est l'expression de droite qui est retournée.

Fonctions anonymes

Certaines fonctions prennent en paramètre une fonction de rappel (callback) qui sera appelée lorsque le traitement de la fonction initiale sera terminé.

Par exemple addEventListener prend un second paramètre la fonction à appeler lorsque l'événement indiqué se produit. Les fonctions setTimeout et setInterval quant à elle appelle une fonction au bout d'un certain temps.

Il est possible de donner le nom d'une fonction existante :

```
function log()
{
    console.log("LOG !");
}

setTimeout(log, 1000);
```

Mais il est également possible de fournir en paramètre une fonction anonyme :

```
setTimeout(function() {
    console.log("LOG !")
```

```
}, 1000);
```

ES6 a apporté une nouvelle syntaxe pour les fonctions anonymes :

```
setTimeout(() => {
    console.log("LOG !")
}, 1000);
```

A priori, rien de révolutionnaire, mais c'est avec les instances de classe que cette syntaxe prend tout son sens, notamment à cause de this.

this

Contrairement à d'autres langages de programmation, this représente le contexte dans lequel une fonction est appelée. Et ce contexte peut changer en fonction d'où est effectué l'appel d'une fonction dans le programme.

Reprenons la classe User précédente avec l'instance user1 :

```
class User
{
    constructor(firstname, lastname)
    {
        this.firstname = firstname;
        this.lastname = lastname;
    }

    logFullName()
    {
        console.log(`${this.firstname} ${this.lastname}`);
    }
}

const user1 = new User("Mélodie", "Durant");
```

Si on appelle la fonction logFullName de user1 :

```
user1.logFullName());

/**
Mélodie Durant
**/
```

Nous obtenons le résultat attendu.

Mais que se passe-t-il si je passe la fonction logFullName de user1 comme fonction de rappelle à setTimeout ?

```
setTimeout(user1.logFullName, 1000);

/**
```

```
undefined undefined
**/
```

Ici, on passe en paramètre à setTimeout une référence sur la méthode logFullName de user1. Mais lorsque la fonction setTimeout appel logFullName, elle ne sait pas que cette dernière doit être appelé dans le contexte de user1. En réalité, setTimeout est une méthode de l'objet window qui contrôle un onglet du navigateur. Lorsque setTimeout appelle logFullName, cette dernière est appelée dans le contexte de window. Or window ne possède pas d'attribut firstname et lastname. Donc quand logFullName tente d'accéder à la valeur de this.firstname, this représentant window et window ne possédant pas d'attribut firstname, undefined est retourné.

Pour palier à ce problème, il existe deux solutions.

1. ES5, la précédente norme de JavaScript, offre une méthode bind permettant de forcer le contexte dans lequel une fonction sera appelée :

   ```
   setTimeout(user1.logFullName.bind(user1), 1000);

   /**
   Mélodie Durant
   **/
   ```

   Ici on indique que logFullName devra être appelé dans le contexte de user1 et non celui de window.

2. ES6, quant à elle, apporte les fonctions fléchées vue précédemment qui maintiennent le contexte d'origine des méthodes appelées :

   ```
   setTimeout(() => {
       user1.logFullName()
   }, 1000);

   /**
   Mélodie Durant
   **/
   ```

   Ici on ne passe plus directement la fonction logFullName de user1, mais une fonction fléchée qui maintiendra le contexte de logFullName dans user1.

   Attention : seules les fonctions anonymes fléchées permettent cela. Une fonction anonyme « ancienne génération » perdra le contexte initial :

   ```
   setTimeout(function(){
       user1.logFullName()
   }, 1000);

   /**
   undefined undefined
   **/
   ```

Conseil : pour les fonctions de rappels, utilisez des fonctions fléchées.