Avant toute chose... quelques rappels

ES6 / ES2015

- ECMAScript 5 est officielle depuis 2009.

  Il s'agit de la norme décrivant la syntaxe JavaScript que l'on connait depuis quelques années.

- ECMAScript 6 est officielle depuis 2015.

  Beaucoup de changements ont été apportés par cette version, qui est en cours d'intégration dans les navigateurs (voir https://kangax.github.io/compat-table/es6/)

- La version 7 (ou 2016) est sortie en 2016.

Les mots-clés `let` et `const` pour nommer les données

```
// `var` limite la portée à la fonction, peu importe les blocs
for (var i=0; i<5; i++) {
  // …
}
console.dir(i); // 5

// `let` limite la portée au bloc
for (let i=0; i<5; i++) {
  // …
}
console.dir(i); // undefined
```

```
const foo = 'bar';
foo = 'test'; // TypeError: Assignment to constant variable
```

Pour des raisons de simplicité, nous utiliserons donc principalement `let` pour définir nos variables lors de cette formation, et `const` pour les constantes.

Les templates literals

```
let name = 'Toto';

const template = `<h1>Bonjour ${name}</h1>
        <p>Vous avez ${18 + 20} ans</p>`;

template; // '<h1>Bonjour Toto</h1>\n\t<p>Vous avez 38 ans</p>'
```

Les litéraux de gabarits ont trois principaux avantages :

1. On peut concaténer et interpoler très facilement ;
2. Les retours à la ligne ne posent plus de problème dans une même chaîne ;

3. On peut les étiqueter pour les traiter rapidement.

```
let mineur = 18;
let majeur = 20;

function tag (strings, ...values) {
  console.log(strings[0]); // '<p>Vous avez '
  console.log(strings[1]); // ' ans</p>'
  console.log(values[0]);  // 38

  return 'Youpi !';
}

tag`<p>Vous avez ${ mineur + majeur } ans</p>`;
```

Attention à la sécurité

Il ne faut pas laisser l'utilisateur manipuler les strings literals, même en partie, sans avoir échappé les données avant.

```
`${console.warn('This is', this)}`; // window{} (not Sparta)
```

Les *arrow functions*

Les fonctions fléchées permettent de définir une fonction de façon raccourcie, tout en évitant de lier certains mots-clés comme `this`, `super` ou `arguments`.

```
let fn = (name, age = 40) => { // Assigner des valeurs par défa
  return `<h1>Bonjour ${name}</h1>
        <p>Vous avez ${age} ans</p>`;
}
fn('Toto', 35);

let fn2 = name => `Bonjour ${name}`; // Lorsqu'il y a un seul a
fn2('Toto');
```

Cette syntaxe raccourcie permet de définir des fonctions anonymes très facilement.

```
fetch('./data.json')
  .then(res => res.json())
  .then(console.table);

fetch('./data.json')
  .then(res => res.json())
  .then((data) => {
        for (let row of data) {
      console.dir(row);
    }
  });
```

Les modules ES6

ES6 permet de fonctionner avec une approche plus modulaire.

Cela permet par exemple de cloisonner chaque classe et valeur, sans définir de variables globales.

Exporter un module

```
export const name = 'Toto';
export var age = 35;

export default class Person {} // Export par défaut
```

L'export par défaut permet plus tard d'importer une valeur sans forcément connaître son nom.

Importer un module

```
import { name, age as ageDuCapitaine } from './test.module.js';
// Importer des valeurs membres d'un module en utilisant un ali

import Test from './test.module.js';
// Importer la valeur par défaut d'un module

import Test, { name } from './test.module.js';
// Importer la valeur par défaut et une valeur nommée
```

Le support sur le terrain... une triste réalité

Node.js supporte (en partie) les modules avec une syntaxe différente depuis des débuts grace à Common.JS.

Peu de navigateurs (pour ne pas dire aucun) supportent les modules ES6 en production (mais ça arrive !).

Mais cela ne veut pas dire qu'il est impossible d'en utiliser pour autant !