programmation web en javascript

5. Fonctions, modules

Rappels sur les fonctions en javascript

rappel

- **rappel : une fonction js est un objet**, donc elle peut être :
  - **créée** via un littéral
  - **affectée** à des variables, tableaux, propriétés
  - passée en **paramètre** à des fonctions
  - retournée comme **résultat** d'une fonction
  - possède des **propriétés** et des **méthodes**

- **et en plus, elle peut être *invoquée* en plaçant des parenthèses: ()**

fonctions en paramètre, et retournée

- Fonction en paramètre, fonction retournée

rappel

```
// une fonction en paramètre :

function calculate(a, b, f) { return f(a,b) ; }

// appeler cette fonction et lui passer 1 argument

calculate(100, 27, (a,b) => a-b ) ;
> 73
```

```
function incrementeur( x ) { return  (a) =>  a+x  ;
}

let incr5 = incrementeur( 5 ) ;

// incr5 est une fonction qui incrémente de 5
> incr5( 5 ) ;
10
```

closures

rappel

- la *closure* d'une fonction : le scope au sein duquel elle est **déclarée**

- la closure est **toujours accessible** lors de l'exécution de la fonction, même si le scope a disparu ou si la fonction est exécutée dans un scope différent !
- la fonction conserve une *référence* vers tous les objets présents dans son scope de déclaration

rappel

scope visible dans la fn résultat

```
function foo(n) {
  let i=1;

  return (x)=> (x + i++)*n;


}


// buzz est une fonction
let buzz = foo(10);

//donc on peut invoquer buzz:
buzz( 5 );
60

buzz( 5 );
70

buzz( 5 );
80
```

appel de la fonction résultat : i et n sont accessibles !

appels successifs : la valeur de i est modifiée

utilisation de closures

- pour *piéger* une variable
- pour créer des variables partagées non globales

rappel

```
function sequence(u0) {
 let u = u0 ;

 return  () =>  u++ ;

}

let next = sequence(10);
> next()
10
> next()
11
> next()
12
> next()
13
```

```
function somme( arr ) {
 let s = 0 ;

 let f = (e)=> {s=s+e ;} ;

 arr.forEach( f ) ;
 return s ;
}

> somme([1,2,3,4]) ;
10
```

invocation immédiate

- une fonction ou une expression de fonction peut être invoquée **immédiatement** à sa création

```
let x = (function return42() {return 42;}) ();

let y = (function () {return 73;}) ();

let z = ((x) =>  1300+x) (37);

x
// 42
y
// 73
z
// 1337
```

limiter les globales grâce à une IIFE

- encapsuler tout le code dans une fonction immédiate

```
let compteur = 0 ;
let i = 1;

let setup=function(x) {
    …
}
function doAll( v ) {
    …
}
function close() {
    …
}

setup( 42 ) ;
doAll( 73 ) ;
close() ;
```

5 variables globales

```
(function () {
  let compteur = 0 ;
  let i = 1;
  let setup=function(x) {
      …
  }
  function doAll( v ) {
      …
  }
  function close() {
      …
  }
  setup( 42 ) ;
  doAll( 73 ) ;
  close() ;
}) () ;
```

0 variable globale

propriétés privées pour un objet

- les objets javascript ne peuvent pas déclarer de propriétés privées
- on se sert de closures

  et d'une IIFE :
  - x, y , origin(): **privées**
  - color, move(), getX(), paint() : interface publique de l'objet

```
let point = (function() {
 let x =0;
 let y =0;
 let origin = ()=>{...} ;

 return {
   color : 'blue',
   move(a,b) {
    x += a ; y += b ;
   },
   getX() {
    return x ;
   },
   paint(c) {
    origin() ;
    this.color= c ;
   }
 }
})() ;
```

Javascript modulaire

bonnes pratiques et patterns de programmation js

- éviter de polluer l'espace global
  - éviter les variables globales en encapsulant dans une IIFE
- utiliser des variables privées dans les objets
  - définir un objet comme le résultat d'une IIFE
- structurer le code
  - utiliser des namespaces

- les modules ES6

le pattern *namespace*

- pas de mécanisme explicite pour structurer l'espace des noms ; on peut se servir des objets :

```
// 6 noms globaux

// constucteurs
function Personne(){}
function Etudiant(){}

//variables
let promo_num = 100 ;
let promo=[ ] ;

//objets
let module1 = {} ;
let module1.data = {
    a:1, b:2};
let module2 = {}
```

```
// NAMESPACE : 1 seul global

let MYAPP = {} ;
MYAPP.Personne = function (){}
MYAPP.Etudiant=function (){}

MYAPP.promo_num = 100 ;
MYAPP.promo=[ ] ;

MYAPP.modules = {}

MYAPP.modules.module1 = {} ;
MYAPP.modules.module1.data =
{a:1,b:2};
MYAPP.modules.module2 = {}
```

Les modules ES6

- ES6 utilise 2 types de scripts :
  - Les scripts classiques `<script>`
  - Les modules `<script type="module">`
- Un module ES6 est un *fichier javascript*
  - toutes les déclarations sont **privées,**
  - sauf celles explicitement **exportées**
  - peut **importer** des déclarations exportées par d'autres modules
  - exécuté une fois chargé

Scripts vs modules

	scripts	modules
balise	`<script>`	`<script type="module">`
mode par défaut	sloppy	strict
déclarations de 1er niveau	globale	Locale au module
valeur de this	`window`	`undefined`
import	non	oui
export	non	oui
extension	`.js`	`.js`

export default

- Mode d'export à utiliser lorsqu'un module exporte 1 seul objet/fonction/variable

m/mod1.js

```
let m = 2 ;
let config = {
    uri: "/le/serveur/de/data"
};

let foo = function(a) {
    return a*m ;
};

let barre = function(a) {
    return a* foo(a);
}

export default {
    conf: config,
    bar: barre
}
```

index.js

```
import m1 from './m/mod1.js' ;

console.log(m1.conf.uri);

console.log(m1.bar(4));
```

- chemin relatif à l'emplacement du fichier qui importe
- seul l'objet exporté est visible

export named

- À utiliser lorsque le module souhaite exporter plusieurs objets distincts, permet de choisir ses imports

m/mod2.js

```
let compteur = 1;

export let foo = function() {
    return ++compteur;
}

let u0=0,u1=1;

export function
fibo_init( u,v ) {
    u0=u; u1=v;
}

export function fibo_next() {
    let u=u0+u1;
    u0=u1;
    u1=u;
    return u;
}
```

index.js

```
import {fibo_init, fibo_next}
       from "./m/mod2.js";
import * as s from "./m/mod2.js";

fibo_init(4,5);
console.log(fibo_next());
console.log(fibo_next());

console.log(s.foo());
console.log(s.fibo_next());
```

- Les noms d'import correspondent aux noms d'export
- On peut importer le module complet en le nommant

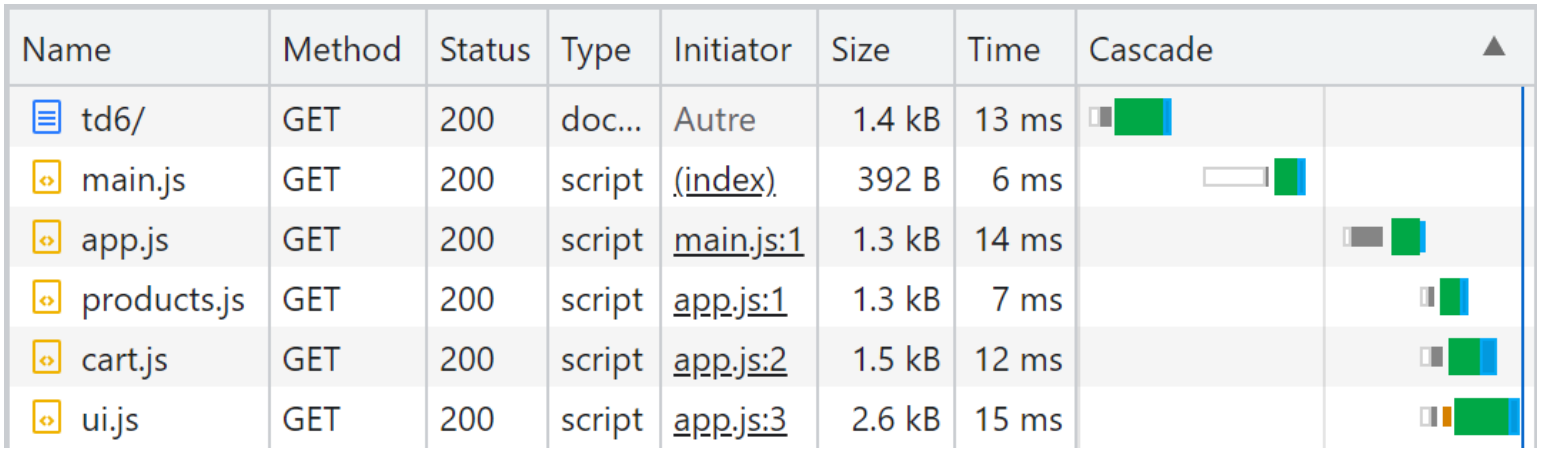
Utilisation dans le navigateur

- **Attention** : les navigateurs chargent les modules uniquement pour les docs avec une url http://
- charger les modules avec `<script type="module">`
- inutile de charger les modules importés, le navigateur les charge automatiquement
  - Les 2 modules m/mod1.js et m/mod2.js sont chargés

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"> <title>modules</title>
</head>
<body>
<h2>un petit test avec les modules ES6 dans le navigateur</h2>

<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>

<script type="module" src="index.js"></script>

</body></html>
```

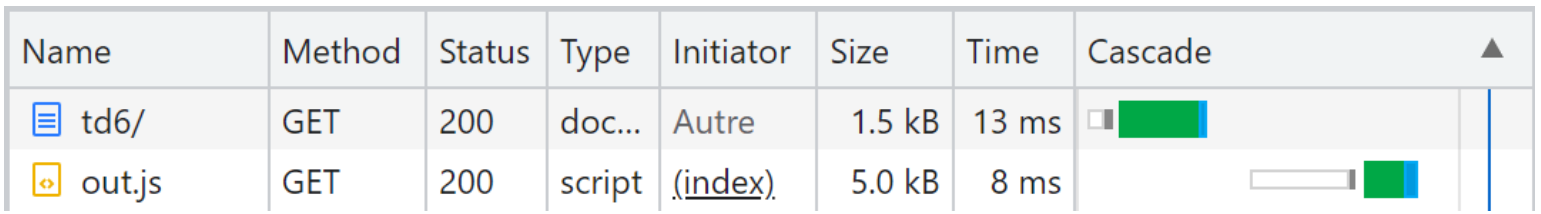
Inconvénient des modules ES6

- Enchaînement de plusieurs requêtes en cascade
  = temps de chargement allongé



Name	Method	Status	Type	Initiator	Size	Time	Cascade
td6/	GET	200	doc...	Autre	1.4 kB	13 ms	
main.js	GET	200	script	(index)	392 B	6 ms	
app.js	GET	200	script	main.js:1	1.3 kB	14 ms	
products.js	GET	200	script	app.js:1	1.3 kB	7 ms	
cart.js	GET	200	script	app.js:2	1.5 kB	12 ms	
ui.js	GET	200	script	app.js:3	2.6 kB	15 ms	

- Solution : rassembler les modules dans un seul fichier
  = utiliser un *bundler*



Name	Method	Status	Type	Initiator	Size	Time	Cascade
td6/	GET	200	doc...	Autre	1.5 kB	13 ms	
out.js	GET	200	script	(index)	5.0 kB	8 ms	

Bundler le + simple : *esbuild*

- Installation*

```
> npm install esbuild
```

- Utilisation directe

```
> ./node_modules/.bin/esbuild main.js --bundle --outfile=out.js
```

- Résultat :
  les modules ES6 sont convertis en pattern *module* :

```
(() => { ... })();
```

* : commande complète sur https://esbuild.github.io/

Automatiser la commande *esbuild* (optionnel)

- Scripter la commande dans *package.json* :

```
{
   "scripts" : {
     "build" : "esbuild main.js --bundle --outfile=out.js"
  }
}
```

- Nouvelle commande :

```
> npm run build
```

Minifier le code avec *esbuild* (optionnel)

- L'option *--minify* compacte le code en renommant les variables/fonctions, en supprimant espaces blancs et commentaires, en réécrivant la synthaxe ...

```
> esbuild main.js --bundle --minify
```

- Cela permet de réduire la taille
  mais aussi d'obfusquer le code
  (c-à-d le rendre "illisible" pour le protéger) :
  s'utilise en production, pas en développement
- (Le code final peut aussi être placé dans la balise <*script*> pour n'avoir qu'un seul fichier *html+js*.)