



JS

programmation web
en javascript
2 / 6

les fonctions en javascript

- **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 *invquée* en plaçant des parenthèses: ()**

fonctions en paramètre, callback

- une fonction est un objet et donc peut être passée en paramètre

```
function add ( a,b ) {return a+b ; }
```

```
let s = (a,b) => a+b ;
```

```
// une fonction en paramètre :
```

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

```
// appeler cette fonction et lui passer 1 argument fn
```

```
> calculate(20,22,add) ;
```

```
> 42
```

```
> calculate(50,23,s) ;
```

```
> 42
```

```
// une expression comme argument
```

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

```
> 73
```

fonctions comme résultats

- une fonction est un objet et peut donc être retournée comme **résultat** d'une fonction

```
function incrementeur( x ) {  
    return (a) => { return a+x ; } ;  
}  
  
let incr5 = incrementeur( 5 ) ;  
  
// incr5 est une fonction qui incrémente de 5  
  
> incr5( 5 ) ;  
10  
> incr5( 12 ) ;  
17
```

scope des variables

- une fonction ou une variable peuvent être déclarées :
 - à tout moment
 - dans n'importe quelle fonction
- **le scope (portée) d'une variable :**
 - **déclarée avec var : toute la fonction** dans laquelle elle est déclarée, ainsi que les fonctions et block imbriqués
 - **déclarée avec let : de la déclaration à la fin du block** de déclaration, ainsi que les fonctions et block imbriqués
- le contexte global (window dans un browser) agit comme un block contenant tout le code : scope global

scope des fonctions

- le **scope** d'une fonction **déclarée** : **tout le block** de déclaration, ainsi que les fonctions et block imbriqués
- les références en avant sont possibles

```
function bar() {  
    console.log('foofoo(1337):', foofoo(1337)); // OK  
  
    function foofoo(x) {return x+42 ;}  
  
    {  
        console.log('a1:', a); // Reference Error  
        console.log('b1:', b); // undefined  
  
        let a = 10;  
  
        console.log('foobuzz(6):', foobuzz(6));  
  
        function foobuzz(x) {  
            console.log('foofoo(100):', foofoo(100));  
            return x + a;  
        }  
  
        var b=8;  
        console.log('foobuzz(3):', foobuzz(3));  
    }  
    console.log('a2:', a); // Reference Error  
    console.log(foobuzz(5)) // Reference Error  
    console.log('b2:', b);  
    console.log('foofoo(42):', foofoo(42));  
}
```


closures

- 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

scope
visible
dans
la fn
résultat

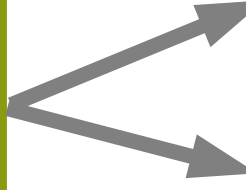
```
function foo(n) {  
  let i=1;  
  
  return (x)=> (x + i++)*n;  
}
```

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



```
// buzz est une fonction  
let buzz = foo(10);  
  
//donc on peut invoquer buzz:  
buzz( 5 );  
60
```

appels successifs :
la valeur de i est
modifiée



```
buzz( 5 );  
70  
  
buzz( 5 );  
80
```

utilisation de closures

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

```
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
```