programmation web en javascript

4 / 6

les objets classiques

- pas de classes ! **ES5**
- ajout dynamique de propriétés à tout moment

```
let voiture = {
        couleurs : {
                car : "rouge" ,
                siege :"marron"
              },
        puissance : 42,
        marque : "tata motors",
        modele : "Manza Sedan"
  } ;
let personne = { } ;
personne.nom = 'joe' ;
```

```
// accès aux propriétés :

let n = personne.nom ;
let m = voiture[ "marque" ] ;
let c = voiture.couleurs.car ;


>"la belle " + m + " " +c + " de "+ n
"la belle tata motors rouge de joe"
```

méthodes : propriétés dont la valeur est une fonction, this

```
let voiture = {

  marque        : "tata motors",
  moteurTourne : false,
  vitesse       : 0,

  demarrer   : function { this.moteurTourne = true, },

  accelerer : function { if (this.moteurTourne)
                         this.vitesse +=x ; },

  repeindre : function { this.couleur = c;}
} ;

voiture.demarrer() ;
console.log(voiture.moteur) ; // → true

voiture.accelerer(20) ;
```

ES5

Une méthode est une propriété d'un objet dont la valeur est une fonction

notation ES6 pour les méthodes

ES6

```
let voiture = {

  marque        : "tata motors",
  moteurTourne : false,
  vitesse       : 0,

  demarrer()    { this.moteurTourne = true, },

  accelerer(x)  { if (this.moteurTourne)
                        this.vitesse +=x ; },

  repeindre(c)  { this.couleur = c;}
} ;

voiture.demarrer() ;
console.log(voiture.moteur) ; // → true

voiture.accelerer(20) ;
```

constructeurs

- un objet peut être créé avec un constructeur

```
let v = new Object() ;    // Équivalent à v = {}

v.couleurs = 'rouge' ;
v.marque='Ferrari' ;
v.puissance=420 ;

v.proprio= {prenom :'joe', nom : 'bar' } ;
```

```
function Voiture(c,m,p) {

      this.couleur = c ;
      this.marque=m ;
      this.puissance=p;
      this.vitesse=0 ;
}

let b = new Voiture('grise','bugatti', 300) ;
let a = new Voiture('bleu','alpine',  130) ;
```

constructeur et méthodes

- définir une méthode dans un constructeur
  - comme une propriété de **this :** elle est dupliquée dans tous les objets créés avec le constructeur – **A EVITER**
  - comme une propriété du ***prototype*** du constructeur : tous les objets créés avec le constructeur ***héritent*** des propriétés du prototype
  - **attention** : ne pas utiliser la notation `()=>{ }` pour définir une méthode dans un prototype si elle utilise **this**

```
function Voiture(c,m,p) {
      this.couleur = c ;
      this.marque=m ;
      this.puissance=p;
      this.vitesse=0 ;

      this.freiner = function() {    // A EVITER
        if (this.vitesse > 1)       // DUPLICATION de la
         this.vitesse -= 1 ;        // METHODE à chaque new
}

Voiture.prototype.accelerer = function( dv ) {
            this.vitesse += dv;
} ;


let royale = new Voiture("rouge", 'bugatti', 300) ;
let a110 = new Voiture("bleu", 'alpine", 120) ;
royale.accelerer( 20 )
```

quelques objets prédéfinis (natifs)

- l'objet global
  - les fns et variables globales sont des propriétés de l'objet global : `window` (navigateur) `global` (node.js)
- l'objet `Math`
  - contient des méthodes pour l'arithmétique et les calculs numériques
- l'objet `Date`
  - manipulation de dates

```
var v = "globale" ;

v === window.v
true
```

```
Math.log10(100)
2

Math.PI
3.141592653589793

Math.round( Math.PI )
3
```

```
let now = new Date() ;

now.getMonth()
0 // janvier
now.getDay()
1 // lundi
```

la valeur de this, le contexte

- Dans la suite, on considère la déclaration de fonction suivante, qui utilise **`this`**

```
function ninja(n, a) {
  this.alias = n ;
  this.age = a ;
}
```

invocation comme fonction

- lorsque `ninja()` est invoquée comme une fonction classique :
- en mode strict : **`this === undefined`**
- en mode sloppy : **`this === window`** (dans un browser), objet global, dont les propriétés sont les variables globales
- la fonction va donc créer des variables globales

```
ninja( "don-san", 42 ) ;

>age
42
>this.age
42
>window.age
42
>var contexte_global = this ;
>contexte_global.age
42
```

invocation comme méthode

- lorsque `ninja()` est invoquée comme une méthode : **`this === objet courant`**

```
let turtle = {
      nom: "donatello",
      init: ninja,
      check() {if (this.age < 40) return "young" ; }
} ;

// invocation comme une méthode :
turtle.init("don-san", 38) ;
turtle.check() ;
"young"

turtle.nom   ;                    // "donatello"
turtle.alias ;                    // "don-san"
turtle.age ;                      // 38
```

invocation comme constructeur

- Toute fonction invoquée en étant précédée du mot clé **`new`** agit comme un constructeur et retourne un nouvel objet
- dans ce cas : **`this=== nouvel objet`**

```
let leonardo = new ninja("leo", 42) ;

console.log(leonardo.alias );
// leo
console.log(leonardo.age );
// 42
typeof leonardo
"object"
```

this dans un callback (I)

- dans un *callback* défini avec *`function`*, **this** est **perdu**

ES5

```
var don= {
    name: 'donatello',
    init() { setTimeout( function() {
                        ninja('don-san', 38)  // this perdu
                }, 300) }                     // dans ninja
   }

var dan= {
    name: 'donatello',
    init() { setTimeout( () => {
                        ninja('don-san', 38)  // this perdu
                }, 300)                        // dans ninja
    }
don.init()
Uncaught TypeError: Cannot set property 'alias' of undefined

dan.init()
Uncaught TypeError: Cannot set property 'alias' of undefined
```

this dans un callback (II)

- dans un callback défini avec `()=>{ }`, this est conservé de manière lexicale

ES6

```
var dona= {
    name: 'donatella',
    init() { setTimeout( () => {
                this.alias='dona';      // this conservé
                this.age=32;            // = objet courant
            }, 300)
    }
}
console.log(dona) ;
// {name: "donatella", init: ƒ}

dona.init()

console.log(dona) ;
//{name: "donatella", alias: "dona", age: 32, init: ƒ}
```

contexte explicite

- toute fonction peut-être invoquée en lui transmettant une **valeur explicite** pour **this**
- l'invocation est faite avec des méthodes de l'objet `Function` :
  - `Function.prototype.call( t, a1, a2..)` : invoque la fonction avec **t** comme valeur de this, et les arguments a1, a2 …
  - `Function.prototype.apply( t, [ a1, a2 ..] )` : invoque la fonction avec **t** comme valeur de **this**, et les arguments a1, a2 … fournis sous forme d'un tableau
  - `Function.prototype.bind( t, a1, a2 ..)` : retourne une ***nouvelle fonction*** ayant **t** comme valeur de **this,** et les arguments a1, a2 .. déjà transmis

```
let raph = {},
    michelangelo = {},
    splinter = {};

ninja.apply(raph, ['raph', 32]);
console.log(raph.alias);
// raph

ninja.call(michelangelo, 'mike', 34);
console.log(michelangelo.age);
// 34


let ninjaSplint = ninja.bind(splinter);

ninjaSplint('hamato', 44);
console.log(splinter.alias);
//hamato
```

exemple d'utilisation d'un contexte explicite

- extraction de méthodes :

```
let p1 = { x : 0,
           y : 0,
         move(a,b) {
                this.x += a ;
                this.y += b ;
         } };

let p2 = { x : 5, y : 5 } ;
```

- **Comment appliquer la méthode `move()` a l'objet p2 ?**

```
let p1 = { x : 0,
    y : 0,
    move(a,b) {   this.x += a ; this.y += b ;
    } };

let p2 = { x : 5, y : 5 } ;

p1.move.call(p2, 4,6);
console.log(p2);
// {x: 9, y: 11}

let moveP2 = p1.move.bind(p2) ;
moveP2(10,10);
console.log(p2);
// {x: 19, y: 21}

p2.move = p1.move.bind(p2);
console.log(p2);
// {x: 19, y: 21, move: f}

p2.move(100,100);
console.log(p2);
// {x: 119, y: 121, move: f}
```

perte de **this** dans un callback

```
let jane = {
    name : 'jane', age : 32,
    friends : ['tarzan', 'cheeta'],
    messageToFriends() {
        this.friends.forEach( function(f) {
                    console.log(this.name+' loves '+ f) ;
                } ) ;
    }
} ;

jane.messageToFriends();
TypeError: Cannot read property 'name' of undefined
```

un contexte pour les callbacks

- solutions **ES5**

```
// en utilisant une closure                                    ES5
messageToFriends() {
    let that=this ;
    this.friends.forEach( function(f) {
        console.log(that.name+' loves '+ f) ;
    } ) ;
}

// en utilisant un contexte explicite
messageToFriends() {
    this.friends.forEach(function (f) {
        console.log(this.name + ' loves ' + f);
    }.bind(this));
}
```

- solution **ES6** : utiliser une fonction flèche

```
let jane = {
    name: 'jane', age: 32,
    friends: ['tarzan', 'cheeta'],
    messageToFriends() {
        this.friends.forEach( (f) => {
            console.log(this.name + ' loves ' + f);
        });
    }
};
jane.messageToFriends();
// jane loves tarzan
// jane loves cheeta
```