# DÉVELOPPEMENT INFORMATIQUE

## LOGIQUE & PROGRAMMATION

### LANGAGE JAVASCRIPT

Y. DELVIGNE

CH. LAMBEAU

# UN OBJET ESSENTIEL JS NATIF :

# ARRAY (tableau)

# 1. POURQUOI FAIRE ?

## GÉRER UN ENSEMBLE (HOMOGÈNE ?) DE DONNÉES

## SANS DEVOIR DÉCLARER UN ENSEMBLE DE VARIABLES

```
// exemple : gérer un ensemble de cotes (0 .. 20)
// pour en faire la moyenne,
// compter le nombre d'échecs, etc ...
var c1 = 12.5, c2 = 19.0;
var c3 = 11.4, c4 = 8.5;
// etc ...
var c29 = 11.5, c30 = 14.5;
var nbEch = 0;
var moy = (c1 + c2 + c3 + ... + c29 + c30) / 30;
if (c1 < 10) nbEch++;
// etc ...
if (c30 < 10) nbEch++;
// et s'il y en avait eu 1000 ? ☹
```

## 2. COMMENT FAIRE ?

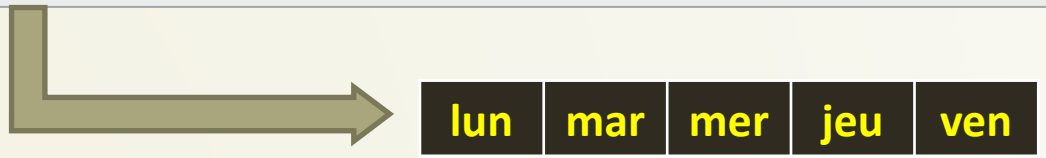
### UTILISER L'OBJET ARRAY :

### (UN SEUL NOM, PLUSIEURS VALEURS ...)

- ACCESSEUR [] (uniquement)
- PROPRIÉTÉS notion d'index, 'longueur'
- MÉTHODES ... (nombreuses)
- ITÉRATION (boucle)

### ❑ INSTANCIATION VIA CONSTRUCTEUR ARRAY()

```
var cotes = new Array();    // tableau vide
var sem = new Array('lun','mar','mer','jeu','ven');
```



## ❑ ENSEMBLE ORDONNÉ DE VALEURS : CONCEPT D'INDEX

```
var sem = new Array('lun','mar','mer','jeu','ven');
```

| lun | mar | mer | jeu | ven |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

c'est la propriété

CHAQUE VALEUR POSSÈDE UNE POSITION (INDEX) COMPTÉE À PARTIR DE 0

## ❑ ACCÈS AUX VALEURS : L'OPÉRATEUR [] (ACCÈS DIRECT)

### ➢ EN 'LECTURE'

```
console.log(sem[0]); // affiche 'lun'
console.log(sem[4]); // affiche 'ven'
```

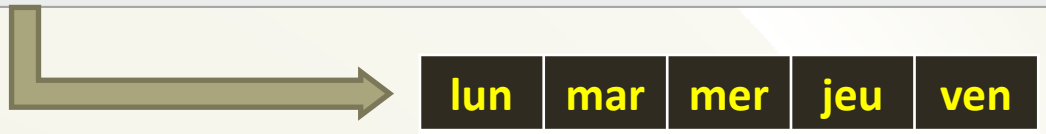
### ➢ EN 'ÉCRITURE'

```
var we = new Array(); // créer tableau vide
we[1] = 'dim';
we[0] = 'sam';
```

| sam | dim |
|---|---|
| 0 | 1 |

## ❑ INSTANCIATION VIA LITTÉRAL : ÉLÉGANCE & CONCISION

```
var cotes = [];        // tableau vide
var sem = ['lun', 'mar', 'mer', 'jeu', 'ven'];
```




## ❑ NOMBRE D'ÉLÉMENTS : PROPRIÉTÉ LENGTH

```
var cotes = [8.5, 11.5, 16, 10.5, 14, 9, 18, 8];
console.log(cotes.length);      // affiche 8
```

**le dernier élément a pour index (position) : length - 1**

## ❑ STRUCTURE DYNAMIQUE : AJOUT OÙ ET QUAND ON VEUT

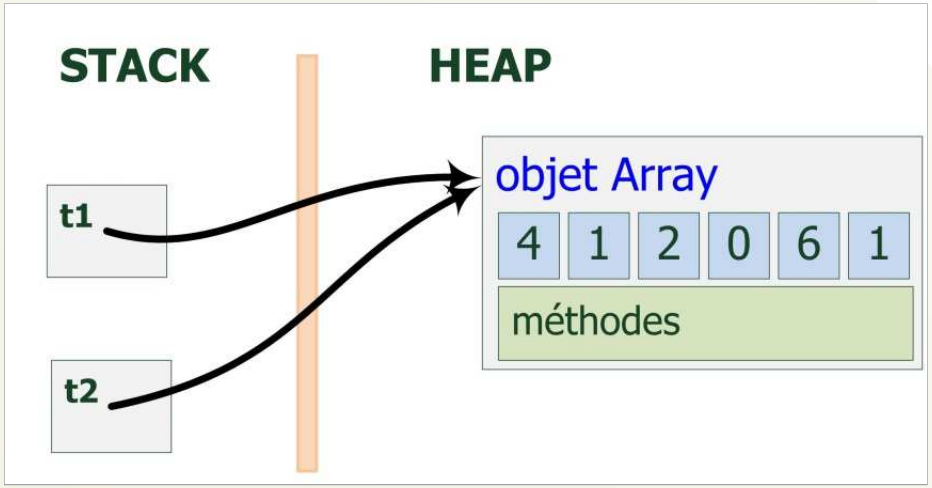
```
cotes[9] = 13.5; cotes[8] = 14;
```

**les éléments manquants sont : undefined**

```
cotes[11] = 20;
```
**➔ cotes[10] == undefined**

## ❑ ARRAY : UN OBJECT → donc les var sont des références !

```
var t1 = [4, 1, 2, 0, 6, 1];
var t2 = t1;  // !!!!!!!!
```




**Comment copier ?**

```
var t1 = [4, 1, 2, 0, 6, 1];
var t2 = new Array(t1);
var t3 = t1.slice()
```

# ❑ ARRAY : UN OBJECT ?? (1)

- ON A DÉFINI PRÉCÉDEMMENT LA COMPOSANTE DONNÉES D'UN OBJET COMME UN ENSEMBLE DE COUPLES (PROPRIÉTÉ : VALEUR)

```
var pers = {
   nom    : 'dugenou',
   prenom : 'pierre',
   age    : 40,
   estBis : false
};
```

valeur = littéral primitif (par exemple)

propriété = identificateur

- ACCÈS AU MOYEN DES NOMS (DE L'OBJET ET DES PROPRIÉTÉS) ET DE L'ACCESSEUR .

```
console.log(pers.nom);
pers.age++;
pers['prenom'] = 'Jules'
```

## ❑ ARRAY : UN OBJECT ?? (2)

```
var t = [4, 1, 2, 0, 6, , 1];
```

- DANS UN ARRAY, OÙ SONT CES COUPLES ?

  IL Y A DES VALEURS, MAIS PAS DE PROPRIÉTÉS ??

  LES PROPRIÉTÉS SONT 'SPÉCIALES' : C'EST LA POSITION (L'INDEX) COMPTÉ À PARTIR DE 0

```
var t = {
   0 : 4,
   1 : 1,
   2 : 2,
   3 : 0,
   4 : 6,
   5 : undefined
   6 : 1
};
```

propriété = identificateur
donc son nom ne peut pas
commencer par un chiffre

COMME ON NE PEUT PAS ÉCRIRE

```
t.0 = 4;
```

ON UTILISE L'ACCESSEUR [ ]

```
t[0] = 4;
```

propriété — valeur

## ❑ STRUCTURE NON TYPÉE : N'IMPORTE QUELLE VALEUR

```
var tab = [];                 // création
tab[0] = 125;                 // typeof tab[0] : "number"
tab[1] = 'hello';             // typeof tab[1] : "string"
tab[2] = false;               // typeof tab[2] : "boolean"
tab[3] = ['yes', 'no'];       // typeof tab[3] : "object"
alert(tab[3][0]);             // affiche 'yes'
```

## ❑ STRUCTURE - TRÈS - DYNAMIQUE

```
var a = 5;                    // "number"
var b = 'hello';              // "string"
var c = ['js', 8.5];          // "object"
var t = [a, b, c];            // !! valeurs des variables !!
```

## ❑ PARCOURIR LA STRUCTURE : NOTION D'ITÉRATEUR

- UN ITÉRATEUR EST UNE BOUCLE QUI 'VISITE' CHAQUE ÉLÉMENT DU TABLEAU VIA SON INDEX (POSITION)
- IL UTILISE DONC UNE VARIABLE ENTIÈRE
  - DÉCLARÉE ET INITIALISÉE À **0** (PREMIER INDEX)
  - LIMITÉE À LA PROPRIÉTÉ LENGTH **- 1** (DERNIER INDEX)
  - INCRÉMENTÉE DE **1** À CHAQUE 'TOUR' (INDEX SUIVANT)

L'ITÉRATEUR **for** EST UNE ÉCRITURE COMPACTE DU **while**

```
var sem = ['lun', 'mar', 'mer', 'jeu', 'ven'];
for (var jr = 0; jr < sem.length; jr++) {
  console.log(jr, sem[jr]);
}
```

## ❑ PARCOURIR LA STRUCTURE : NOTION D'ITÉRATEUR

- ITÉRATEUR 'SPÉCIAL' POUR LES OBJETS EN JAVASCRIPT
- ÉQUIVALENT AU FOR 'CLASSIQUE', ÉCRITURE COMPACTE
- UN PEU MOINS PERFORMANT (RAPIDITÉ)

**for (**«property» **in** «object»**) { //** code **}**

ICI L'OBJET EST LE TABLEAU ET LA PROPRIÉTÉ L'INDEX (POSITION)

```
var sem = ['lun', 'mar', 'mer', 'jeu', 'ven'];
for (var jr in sem) {
  console.log(jr, sem[jr]);
}
```

## ARRAY

```
// exemple : gérer un tableau de cotes (0 .. 20)
// pour en faire la moyenne,
// compter le nombre d'échecs, etc ...
var tc = [12.5, 19.0, 11.4, 8.5, 11.5, 14.5];
var nbEch = 0, total = 0, moyenne;
for (var i = 0; i < tc.length; i++) {
  total += tc[i];
  if (tc[i] < 10) nbEch++;
}
moyenne = total / tc.length;

// qu'il y ait 2 cotes ou 1000 ! ☺

// ou bien
// for (var i in tc) { ... }
```

## ❑ SOUPLESSE D'UTILISATION : NOMBREUSES MÉTHODES

- **AJOUT D'ÉLÉMENTS** (au début / à la fin)
- **SUPPRESSION D'ÉLÉMENTS** (au début / à la fin)
- **FUSION** (de deux tableaux en un seul)
- **ÉCLATEMENT** (d'un tableau en deux tableaux, ou plus …)
- **RENVERSEMENT** (premier ⇔ dernier, etc...)
- **TRI** (ordonnancement)
- **FILTRAGE** (selon critère)
- **COPIE** (d'un tableau dans un autre)
- *… et d'autres encore …*

# ARRAY : MÉTHODES

| | |
|---|---|
| concat(value1 [, value2,…]) | *Concatenates two arrays and returns the new array thus formed.* |
| every(testFn(element, index, array)) | *Iterates over the array, executing testFn() on every element. Returns true if all iterations return true. Otherwise, it returns false.* |
| filter(testFn(element, index, array)) | *Iterates over the array, executing testFn() on every element. Returns a new array of elements that pass testFn().* |
| forEach(fn(element, index, array)) | *Iterates over the array, executing fn() on every element.* |
| indexOf(element [, startIndex]) | *Returns an index of the specified element if found, or -1 if not found. Starts at startIndex if specified.* |
| join([separator]) | *Joins all the elements of an array into a single string delimited by separator if specified.* |
| lastIndexOf(element [, startIndex]) | *Searches an array starting at last element and moves backwards. Returns an index of the specified element if found, or -1 if not found. Starts at startIndex if specified.* |
| map(fn(element, index, array)) | *Iterates over the array, executing fn() on every element. Returns a new array based on the outcome of fn().* |
| pop() | *Pops the last element from the end of the array and returns that element.* |
| push(value1 [, value2, …]) | *Pushes one or more elements onto the end of the array and returns the new length of the array. The array's new length is returned.* |
| reverse() | *Reverses the order of the elements in the array, so the first element becomes the last and the last becomes the first.* |
| shift() | *Removes the first element from the beginning of the array and returns that element.* |

# ARRAY : MÉTHODES

| | |
|---|---|
| slice(startIndex [, endIndex]) | *Returns a slice of the array starting at the start index and ending at the element before the end index.* |
| some(testFn(element, index, array)) | *Iterates over the array, executing testFn() on every element. Returns true if at least one result of testFn() is true.* |
| sort([sortFn(a,b)]) | *Sorts the elements of the array. Executes sortFn() for sorting if it is provided.* |
| splice(startIndex [, length, value1, ]) | *Removes the amount of elements denoted by length starting at startIndex. Provided values replace the deleted elements. Returns the deleted elements.* |
| toString() | *Converts the Array object into a string.* |
| unshift(value1 [, value2, …]) | *Adds elements to the beginning of the array and returns the new length.* |
| valueOf() | *Returns the primitive value of the array.* |

# ARRAY : EXEMPLES DE MÉTHODES

## ➢ AJOUT/SUPPRESSION D'ÉLÉMENTS

```
var t = [1, 5, 2, 7, -1];  // création préalable
```

### ▪ À LA FIN DU TABLEAU

```
t.push(9, 5);          // ➔ [1, 5, 2, 7, -1, 9, 5]
t.pop();               // ➔ [1, 5, 2, 7, -1, 9]
```

### ▪ AU DÉBUT DU TABLEAU

```
t.unshift(3, 0, 6);  // ➔ [3, 0, 6, 1, 5, 2, 7, -1, 9]
t.shift();           // ➔ [0, 6, 1, 5, 2, 7, -1, 9]
```

ces quatre méthodes modifient le tableau

`push()` et `unshift()` permettent d'ajouter un ou plusieurs éléments et **retournent `t.length`** (nouveau nombre d'éléments)

`pop()` et `shift()` - !! pas de paramètre !! - n'enlèvent que le dernier/premier élément et **retournent cet élément**

# ARRAY : EXEMPLES DE MÉTHODES

➢ **AJOUT/SUPPRESSION (PLUS GÉNÉRAL) : SPLICE()**

```
var t = [1, 5, 2, 7, -1];  // création préalable
```

- **AJOUT : `t.splice(debut, 0, liste_d_elements)`**

```
t.splice(1, 0, 3, 9); // ➔ [1, 3, 9, 5, 2, 7, -1]
```

`splice()` en ajout modifie le tableau
et **ne renvoie rien**

- **SUPPRESSION : `t.splice(debut, nombre_à_supprimer)`**

```
t.splice(2, 2);       // ➔ [1, 5, -1]
```

`splice()` en suppression modifie le tableau
et **renvoie** un tableau des éléments supprimés

**ne pas confondre `splice()` et `slice()`**

# ARRAY : EXEMPLES DE MÉTHODES

## ➢ EXTRACTION/COPIE (D'UN TABLEAU DANS UN AUTRE)

```
var t1 = [1, 5, 2, 7, -1];   // création préalable
var t2 = [];                 // vide
```

### ▪ EXTRACTION : RENVOIE UNE 'TRANCHE' DU TABLEAU

```
t1.slice(0, 3);      // ➔ [1, 5, 2]
t1.slice(3);         // ➔ [7, -1]
t1.slice(-1);        // ➔ [1, 5, 2, 7]
```

`slice()` ne modifie pas le tableau

- 2 paramètres : indice début (compris) et indice fin (non compris)
- 1 paramètre : indice début (compris) jusqu'à la fin
- paramètre négatif : on compte les indices depuis la fin

### ▪ COPIE (PARTIELLE OU TOTALE) : AFFECTER SLICE()

```
t2 = t1.slice(0, 3); // ➔ t2 contient [1, 5, 2]
t2 = t1.slice(0);    // ➔ t2 contient [1, 5, 2, 7, -1]
```

# ARRAY : EXEMPLES DE MÉTHODES

## ➢ CONCATÉNATION (FUSION DE DEUX TABLEAUX)

```
var t1 = [1, 5, 2, 7];  // création préalable
var t2 = [3, 6];
var t3 = [];
```

```
t1.concat(t2);          // ➔ [1, 5, 2, 7, 3, 6]
t2.concat(t1);          // ➔ [3, 6, 1, 5, 2, 7]
t1.concat(0, 4);        // ➔ [1, 5, 2, 7, 0, 4]
t1.concat([0, 4]);      // ➔ [1, 5, 2, 7, 0, 4]
t1.concat(-1, [0, 4]); // ➔ [1, 5, 2, 7, -1, 0, 4]
t3 = t1.concat(t2);     // ➔ t3 contient [1, 5, 2, 7, 3, 6]
```

`concat()` ne modifie pas les tableaux passés en paramètres

- paramètre(s) : un ou plusieurs tableaux, quelle que soit leur nature (objet ou littéral), des valeurs scalaires sont considérées comme des tableaux d'un seul élément

# ARRAY : EXEMPLES DE MÉTHODES

➢ **RENVERSEMENT**

```
var t = [1, 5, 2, 7, -1];   // création préalable
```

```
t.reverse();   // ➔ t contient [-1, 7, 2, 5, 1]
```

reverse() modifie le tableau et renvoie le tableau

➢ **CONVERSION EN "STRING"**

```
var t = [1, 5, 2, 7, -1];   // création préalable
```

```
t.toString();   // ➔ "1,5,2,7,-1"
t.join();       // ➔ "1,5,2,7,-1"
t.join(';');    // ➔ "1;5;2;7;-1"
t.join(' ');    // ➔ "1 5 2 7 -1"
t.join('');     // ➔ "1527-1"
```

ces méthodes ne modifient pas le tableau et renvoient un string

# ARRAY : EXEMPLES DE MÉTHODES

➢ **FILTRAGE : RENVOYER UNE PARTIE DU TABLEAU CORRESPONDANT À UN CRITÈRE (UNE CONDITION)**

```
var t = [1.2, 5.5, 9.0, 7.2, -1.8];  // création
```

***on voudrait les éléments compris dans l'intervalle 5 à 10***

il faut créer une fonction booléenne (prédicat) permettant de vérifier qu'un élément vérifie ou non la condition

```
function verif(x) {
  return (x >= 5.0 && x <= 10.0);
}
```

cette fonction (son nom) est passée comme paramètre à la méthode `filter()` qui l'appliquera à chaque élément du tableau

```
var t1 = t.filter(verif);   // ➔ [5.5, 9.0, 7.2]
```

filter() ne modifie pas le tableau

# ARRAY : EXEMPLES DE MÉTHODES

➢ **TRI : ORDONNANCER LES ÉLÉMENT D'UN TABLEAU**

```
var t = ['lui', 'moi', 'elle', 'eux', 'on', 'je'];
```

simplement (???) invoquer la méthode sort() qui modifie le tableau

```
t.sort(); // ➔ t contient
                ['elle','eux','je','lui','moi','on']
```

**PAS SI SIMPLE !!!**

```
var t = [5, 11, 2, 1, 21, 14, 3];
```

```
t.sort(); // ➔ t contient [1, 11, 14, 2, 21, 3, 5] ?☹?☹?
```

☝ la méthode sort() effectue - par défaut - un tri alphanumérique

➔ les "number" sont d'abord convertis en "string"

# ARRAY : EXEMPLES DE MÉTHODES

**COMMENT EFFECTUER UN TRI NUMÉRIQUE ???**

```
var t = [5, 11, 2, 1, 21, 14, 3];
```

il faut créer une fonction comparatrice permettant de vérifier pour un couple d'éléments leur relation d'ordre (<, ==, >)

par convention, la fonction renvoie :

- une valeur négative (si x < y)
- une valeur nulle (si x == y)
- une valeur positive (si x > y)

```
function compar(x, y) {
  if (x > y) return +1;
  if (x < y) return -1;
  return 0;
}
```

sur les nombres, on écrira plutôt :

```
function compar(x, y) {
  return x - y;
}
```

cette fonction (son nom) est passée comme paramètre à la méthode sort() qui l'appliquera à chaque couple d'éléments du tableau pour les échanger au besoin

```
t.sort(compar); // ➔ t contient [1, 2, 3, 5, 11, 14, 21] ☺
```

# ARRAY : EXEMPLES DE MÉTHODES

## UTILISATION DE FONCTIONS ANONYMES POUR ÉVITER LA CRÉATION DE FONCTIONS 'EXTERNES'

### REVISITONS LE TRI NUMÉRIQUE

```
var t = [5, 11, 2, 1, 21, 14, 3];
t.sort(function(x,y){return x-y;});
```

### REVISITONS LE FILTRAGE NUMÉRIQUE

```
var t = [1.2, 5.5, 9.0, 7.2, -1.8];
t.filter(function(x){return (x>=5 && x<=10);});
```

### 'FAIRE' QUELQUE CHOSE SUR CHAQUE ÉLÉMENT

```
var cotes = [8, 11, 13, 9, 15, 18, 20, 5];
t.forEach(function(x){if (x!=20) x++;});
```

# ARRAY : EXEMPLES DE MÉTHODES

## RECOURIR AU CHAÎNAGE DE MÉTHODES SI NÉCESSAIRE

### LE TRI NUMÉRIQUE : VOIR LE TABLEAU TRIÉ SANS LE TRIER !

```
var t = [5, 11, 2, 1, 21, 14, 3];
console.log(t);
console.log(t.slice().sort(function(x,y){return x-y;}));
console.log(t);
```

une copie !

```
> var t = [5, 11, 2, 1, 21, 14, 3];
  console.log(t);
  console.log(t.slice().sort(function(x,y){return x-y;}));
  console.log(t);
  [5, 11, 2, 1, 21, 14, 3]
  [1, 2, 3, 5, 11, 14, 21]
  [5, 11, 2, 1, 21, 14, 3]
< undefined
```

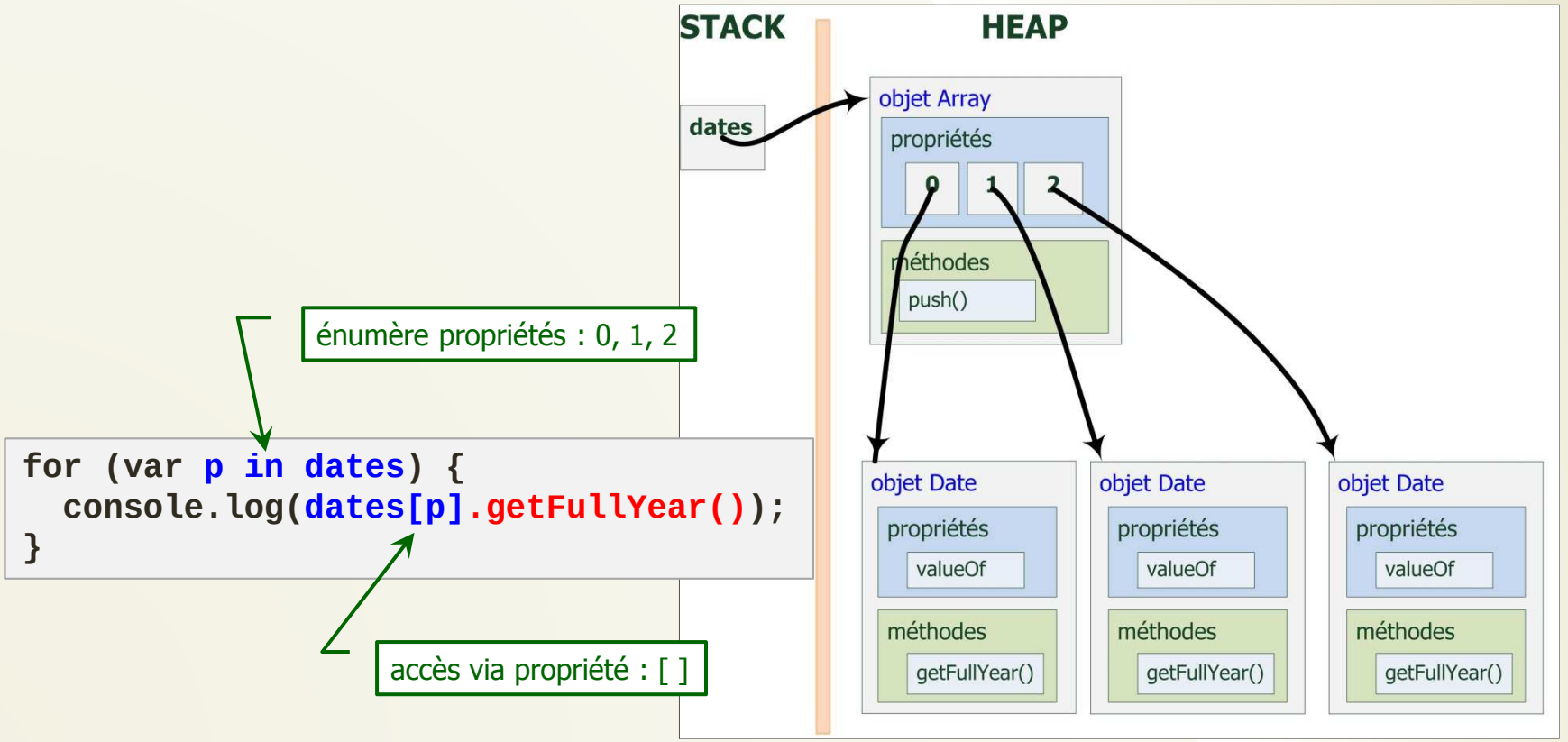
## TABLEAUX D'OBJETS NATIFS SIMPLES : DATES

```
var dates = [];
dates[0] = new Date(2014, 0, 31);
dates[4] = new Date(2010, 6, 14);
dates[1] = new Date(2012, 7, 1);
dates[3] = new Date(2011, 10, 3);
dates[2] = new Date(2013, 3, 17);

for (var d in dates) {
  console.log(dates[d].getFullYear());
}

var tp = [];
for (var d in dates) {
  tp.push(dates[d].getFullYear());
}
console.log(tp.sort(function(x,y) {return x - y}));
```

# ARRAY : TABLEAUX D'OBJETS

## TABLEAUX D'OBJETS NATIFS SIMPLES : DATES

```
var dates = [];
dates.push(new Date(2014, 0, 31));
dates.push(new Date(2012, 7, 1));
dates.push(new Date(2013, 3, 17));
```



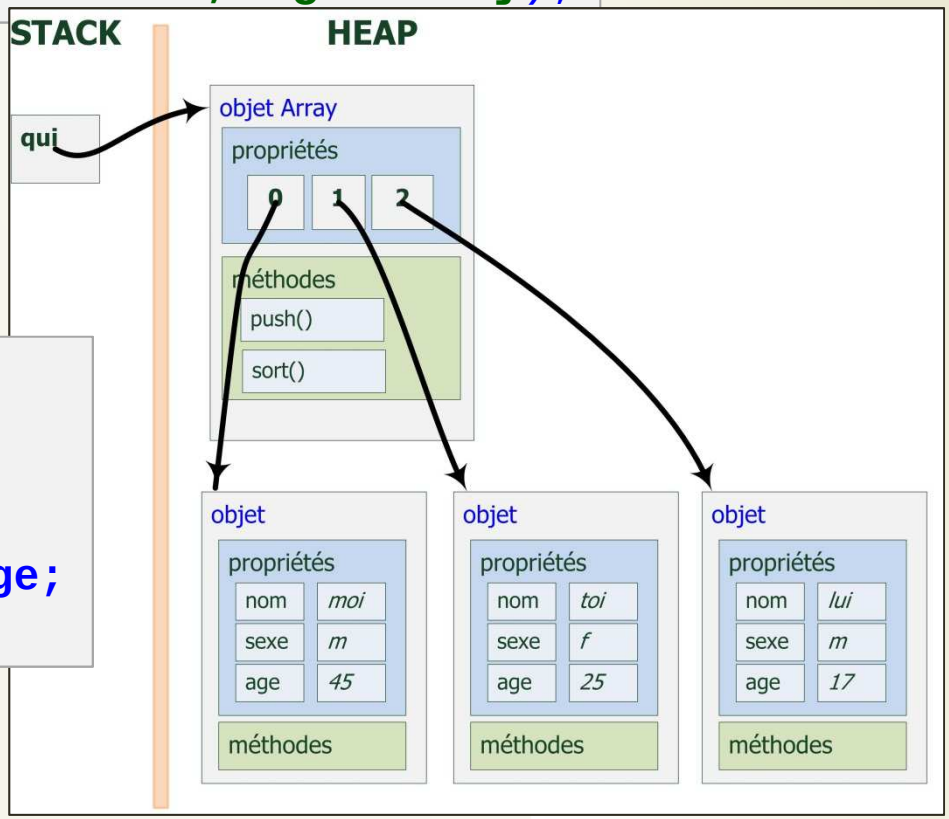
## TABLEAUX D'OBJETS PLUS COMPOSITES

```
var qui = [];
var p = {nom : 'moi', sexe : 'm', age : 45};
qui.push(p);
p = {nom : 'toi', sexe : 'f', age : 25};
qui.push(p);
p = {nom : 'lui', sexe : 'm', age : 17};
qui.push(p);
p = {nom : 'elle', sexe : 'f', age : 39};
qui.push(p);

for (var o in qui) {
  console.log(qui[o].nom);
}

qui.sort(function(x,y) {return x.age - y.age});
```

# ARRAY : TABLEAUX D'OBJETS

## TABLEAUX D'OBJETS PLUS COMPOSITES

```
var qui = [];
qui.push({nom : 'moi', sexe : 'm', age : 45});
qui.push({nom : 'toi', sexe : 'f', age : 25});
qui.push({nom : 'lui', sexe : 'm', age : 17});
```




```
for (var o in qui) {
  console.log(o, qui[o].nom);
}
qui.sort(function(x,y) {
           return x.age - y.age;
         });
```