Node.JS

Rappels JavaScript

Rappels JavaScript
Partie 1

JavaScript : historique

- Langage apparu en 1996 dans Netscape Navigateur 2
  - Copié par Microsoft sous le nom JScript
  - Normalisé avec l'ECMA en 1997 : on parle d'ECMAScript (ES)

- Prévu initialement pour "un peu d'interactivité dans les navigateurs"
  - Effet de rollover (impossible à l'époque en CSS)
  - Vérification de formulaire
  - Source de problèmes très impopulaire

- Devenu populaire à partir de 2004 avec Ajax / le Web 2.0
  - Cheval de bataille des principaux navigateurs
  - Langage parmi les plus populaires
  - Usage bien au-delà des navigateurs (scripting système ou applicatif, langage côté serveur avec Node.JS)

JavaScript : caractéristiques

- Syntaxe inspirée du C
- Dynamique
  - Typage dynamique : pas d'association figée de type aux variables
  - Évaluation du code à l'exécution

- "Functions are first class cityzen" (intraduisible)
  - Fonction = objet utilisable en tant que valeur / paramètre / valeur de retour d'une autre fonction
  - Usage répandu des fonctions anonymes (notamment pour callback)
  - Aucun contrôle du nombre de paramètres

JavaScript : caractéristiques

- Orienté Objet
  - Originellement sans classe : fonctionnement à base de prototype
  - Possibilité d'objet "sans classe"
  - Objet pouvant être vu comme un tableau associatif
  - Dynamique : chaque objet est autonome et peut se voir ajouter/supprimer/modifier des propriétés ou des méthodes
  - Fonctions comme constructeur d'objet et comme méthodes.
- Notation JSON (JavaScript Object Notation)
- Grande capacité d'introspection

JavaScript : points faibles

- Point faibles de JavaScript/ES5
  - Objets "tout en public", pas de private / protected
  - Système d'héritage très particulier
  - Pas de package pour regrouper les classes
  - Pas de possibilité de typage fort ➜ augmente le risque de bugs
  - Langage sans système d'inclusion de fichier (au niveau HTML)
  - Difficile à mettre au point
  - Difficile à maintenir
- Amélioration de certains points en JavaScript/ES6
- Existence de nombreuses techniques pour palier y compris en JavaScript/ES5
- Incontournable jusqu'à aujourd'hui dans les navigateurs
- Dans le futur WebAssembly pourrait changer la donne

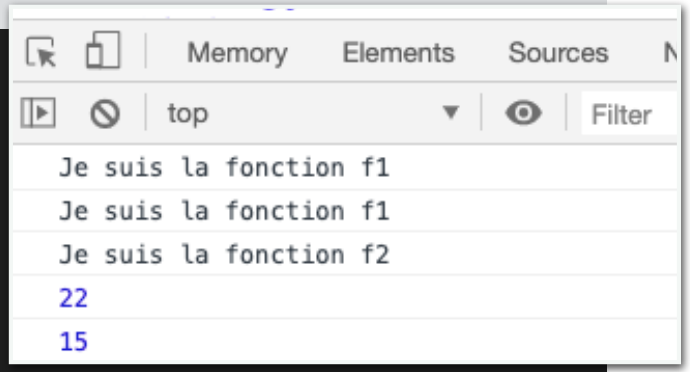
Rappels JavaScript Partie 2

Les fonctions

- Exemple :

/function.js

```
1   // version "traditionnelle"
2   function f1() {
3       console.log("Je suis la fonction f1");
4   }
5   f1();
6   var f1_2 = f1;
7   f1_2();
8
9   // Version plus courante "valeur de variable"
10  var f2 = function() {
11      console.log("Je suis la fonction f2");
12  }
13  f2();
14
15  // Fonction en tant que valeur de retour
16  function generateur(ajout) {
17      return function (val) {
18          return val + ajout;
19      }
20  }
21  var add5 = generateur(5);
22  console.log( add5(17) ); // affiche 22
23  console.log( generateur(10)(5) );
```



- Il existe encore d'autres moyens de déclarer une fonction en JavaScript/ES5
- Fonctions anonymes très utilisées !

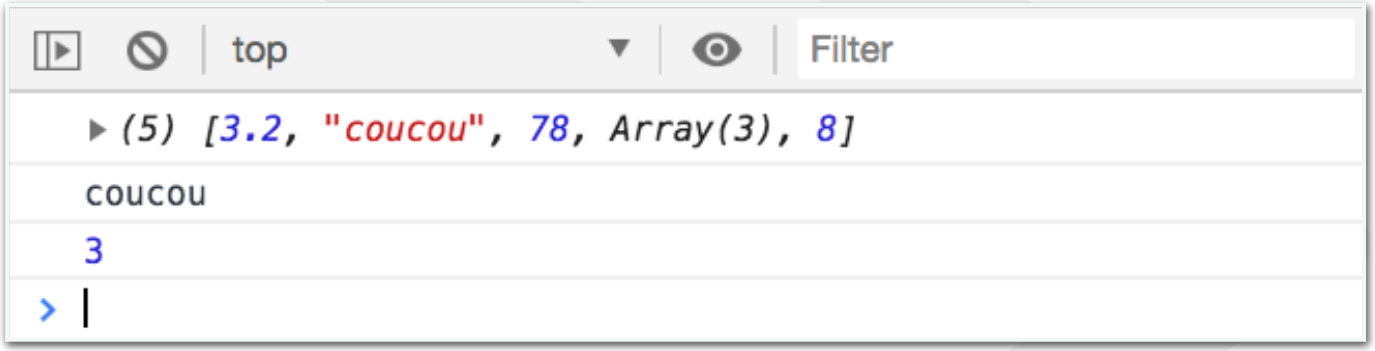
JSON

- Exemple :

/json.js

```
1   var tab = [1,2,"coucou",78,[1,2,3],8];
2
3   console.log(tab);
4   console.log(tab[1]);
5   console.log(tab[3][2]);
```

Tableau en notation JSON



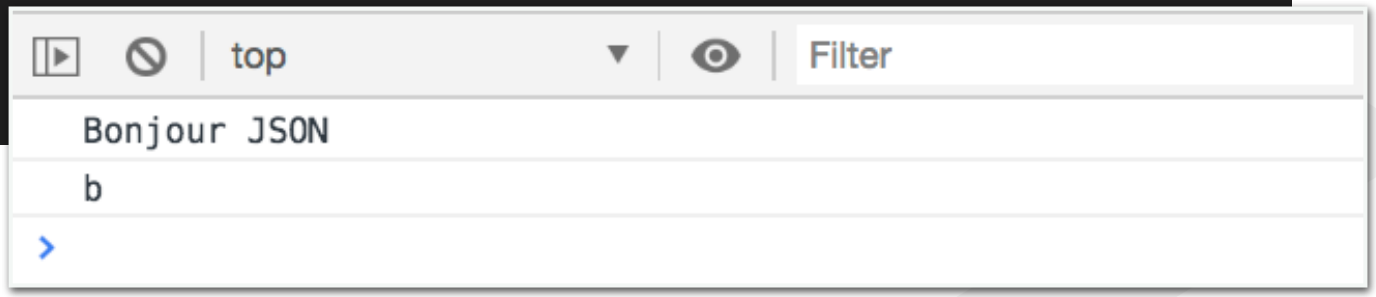
JSON

- Exemple :

/json.js

```
1   var obj = {
2       x : 8,
3       message : "Bonjour JSON",
4       tab : ['a','b','c'],
5
6       methode : function() {
7           console.log(this.message);
8           console.log(this.tab[1]);
9       }
10  };
11
12  obj.methode();
```

Objet anonyme en notation JSON



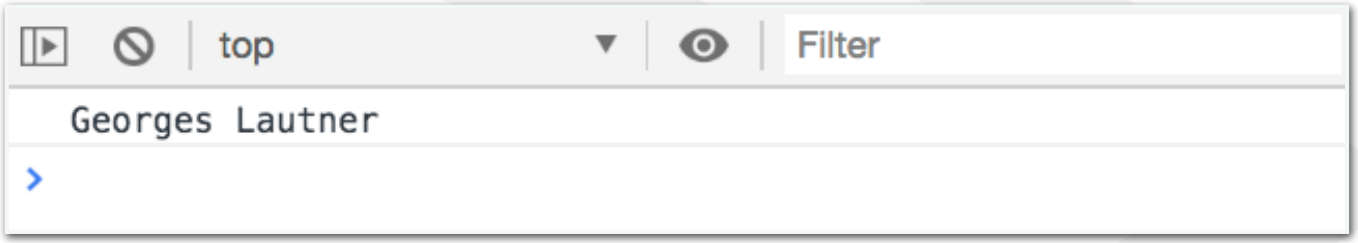
- Les objets anonymes sont très utilisés en JavaScript

JSON

- Exemple :

/json.js

```
1  var data = [
2      {titre:'Le grand bleu',         realisateur:'Luc Besson', sortie:'1988'},
3      {titre:'Les tontons flinguers', realisateur:'Georges Lautner', sortie:'1963'},
4      {titre:'Avatar',                realisateur:'James Cameron', sortie:'2009'}
5  ];
6
7  console.log(data[1].realisateur);
```



- Très facile de créer un ensemble de données en JS

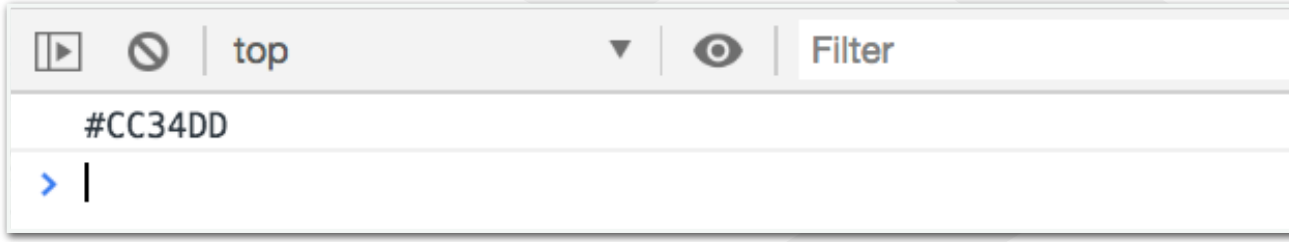
Rappels JavaScript Partie 3

Fonction comme constructeur

- Exemple "classique" :

/rectangle.js

```
1  function Rectangle(x,y,hauteur,largeur,couleur) {
2      this.x = x;
3      this.y = y;
4      this.hauteur = hauteur;
5      this.largeur = largeur;
6      this.couleur = couleur;
7  }
8
9  var rect = new Rectangle(2,7,1,1,'#CC34DD');
10 console.log(rect.couleur);
```



- L'utilisation de this fait que la fonction est un constructeur.

Fonction comme constructeur

- Exemple "plus JavaScript" :

/rectangle.js

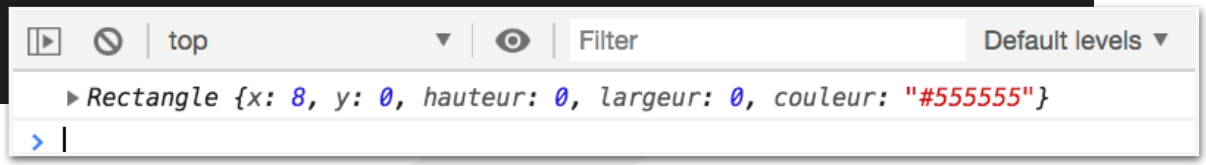
```
1   function Rectangle(config) {
2       this.x = 0;
3       this.y = 0;
4       this.hauteur = 0;
5       this.largeur = 0;
6       this.couleur = '#000000';
7
8       for(var prop in this) {
9           if(config.hasOwnProperty(prop)){
10              this[prop] = config[prop];
11          }
12      }
13  }
14
15  var rect = new Rectangle({
16      couleur : '#555555',
17      x : 8
18  });
19
20  console.log(rect);
```

Un seul paramètre !

Valeur par défaut des propriétés

Boucle sur les propriétés de this
Recherche la même propriété prop dans config
La récupère si elle existe

Ordre des propriétés quelconque



- Utilise un objet en paramètre de la construction
- Autorise de ne pas tout initialiser / initialiser dans un ordre quelconque
- Note : utilise la notation "tableau associatif" des objets JavaScript (ligne 10)

Rappels JavaScript
Partie 4

Portée des variables

- Portée des variables = scope en anglais
- Les blocs n'ont pas de portées en JS, seulement les fonctions.

- Important : accéder à une variable qui n'existe pas lance une exception ReferenceError

/portee.js

```
1   console.log(nom);
```

```
▶ Uncaught ReferenceError: nom is not defined          06-portee-undefined.js:1
    at 06-portee-undefined.js:1
>
```

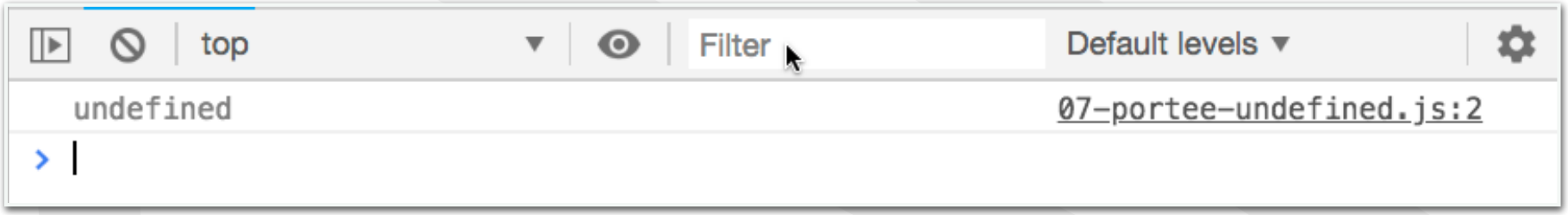
- Comprendre ReferenceError = variable non déclarée

Portée des variables

- Possibilité de déclarer une variable sans affecter de valeur ➜ `undefined`

/portee.js

```
1   var nom;
2   console.log(nom);
```



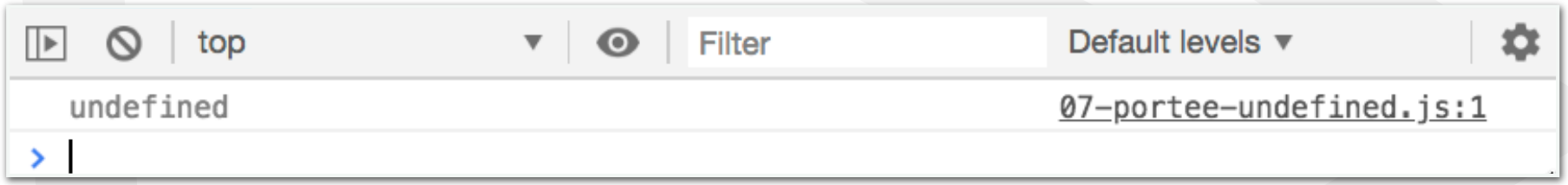
- `undefined` est une valeur particulière et qui peut être testée
- `undefined` ≠ `ReferenceError`
- Déclarée et définie sont deux notions différentes.

PORTÉE DES VARIABLES : HOISTING

- Hoisting (remontée des déclarations de variables)
  - Remontée automatique des déclarations de variable en haut du bloc courant (scope courant, au niveau de la fonction)
  - Comportement singulier et source de problèmes

/hoisting.js

```
1   console.log(nom);   // ReferenceError attendu
2   var nom;
```



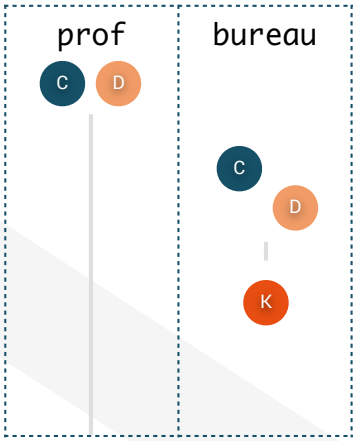
- Bien qu'écrit après le `console.log()`, la déclaration `var nom` est traitée avant à cause du hoisting.

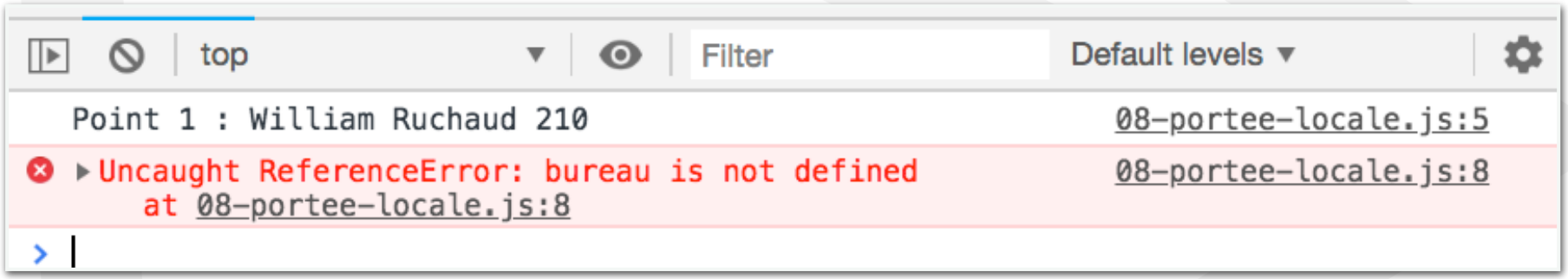
PORTÉE DES VARIABLES

- Portée des variables : fonction dans laquelle elles sont déclarées.

/portee.js

```
1  var prof = "William Ruchaud";
2
3  function afficheBureau() {
4      var bureau = 210;
5      console.log("Point 1 : "+prof+" "+bureau);
6  }
7
8  afficheBureau();
9  console.log("Point 2 : "+prof+" "+bureau);
```





- Ligne 4 : `bureau` n'est disponible que pour la fonction `afficheBureau()` car définie à l'intérieur

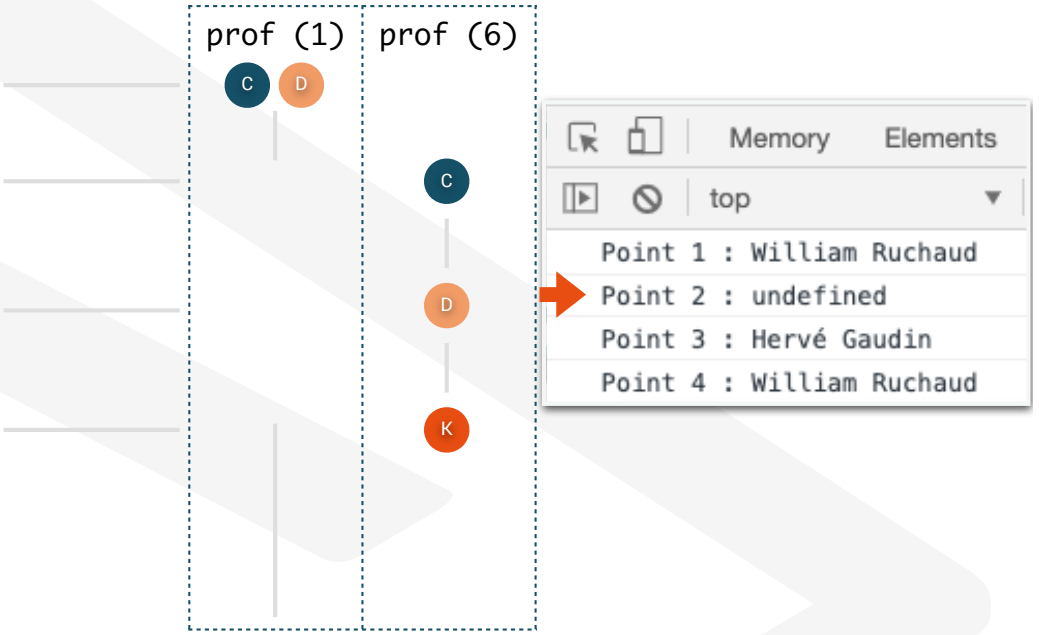
C Création D Définition K Destruction

Portée des variables

- Possibilité de masquage par une variable locale

/masquage.js

```
1   var prof = "William Ruchaud";
2
3   function profDeRobotique() {
4       console.log("Point 2 : "+prof);
5
6       var prof = "Hervé Gaudin";
7       console.log("Point 3 : "+prof);
8   }
9
10  console.log("Point 1 : "+prof);
11
12  profDeRobotique();
13  console.log("Point 4 : "+prof);
```



- Point 1 : normal, `prof` définie ligne 1
- Point 2 : hoisting de la variable `prof` définie ligne 6 (comme si `var prof;` était exécuté au tout début de la fonction)
- Point 3 : affectation de la variable locale `prof`, effet de masquage de celle définie ligne 1
- Point 4 : en dehors de la fonction seule `prof` définie ligne 1 existe.

Portée des variables

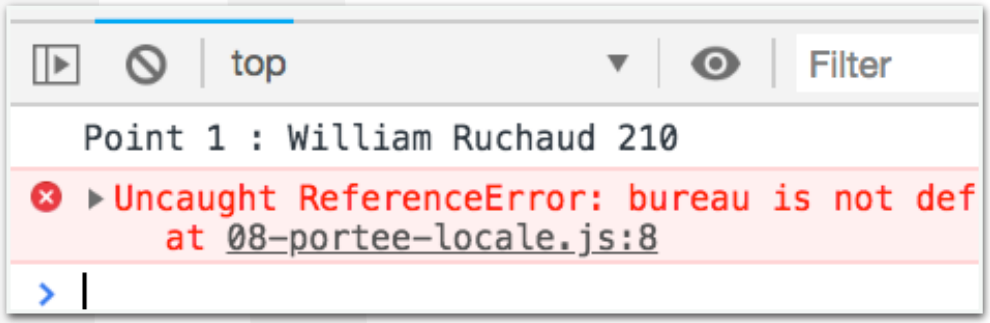
- Déclaration automatique à l'affectation (sans `var`)
- Véritable danger car source de bugs

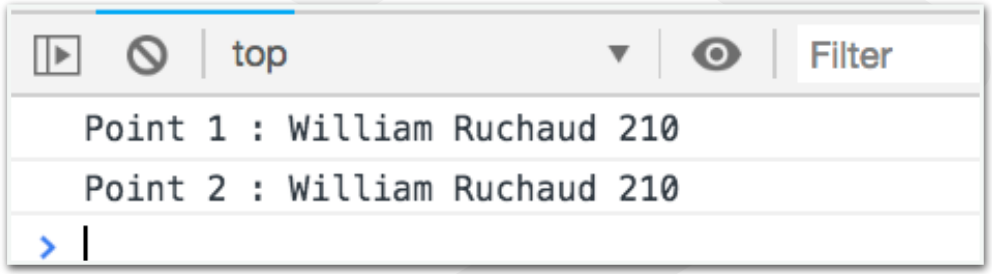
/var.js

```
1  var prof = "William Ruchaud";
2
3  function afficherBureau() {
4      var bureau  = 210;
5      console.log("Point 1 : "+prof+" "+bureau);
6  }
7  afficherBureau();
8  console.log("Point 2 : "+prof+" "+bureau);
```

/sans-var.js

```
1  var prof = "William Ruchaud";
2
3  function afficherBureau() {
4      bureau  = 210;
5      console.log("Point 1 : "+prof+" "+bureau);
6  }
7  afficherBureau();
8  console.log("Point 2 : "+prof+" "+bureau);
```





- Sans `var`, `bureau` a une portée globale, avec `var` une portée locale
- Risque par simple faute de frappe lors d'une affectation de déclarer une nouvelle variable (peut-être supposée locale)

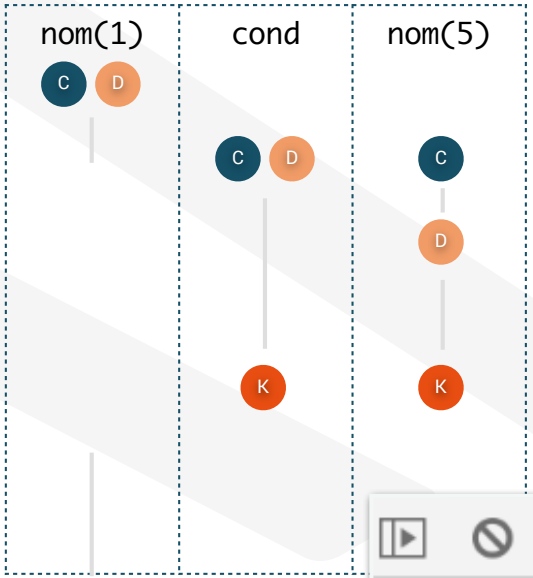
Rappels JavaScript Partie 5

PORTÉE DES VARIABLES

- Piège du hoisting

/hoisting.js

```
1   var nom = "William Ruchaud";
2
3   function demo(cond) {
4       if(cond) {
5           var nom = "Hervé Gaudin";
6       }
7       return nom;
8   }
9
10  console.log("Point 1 : "+demo(false));
11  console.log("Point 2 : "+demo(true));
12  console.log("Point 3 : "+nom);
```



- Point 1 : la variable définie ligne 5 prend le pas sur celle définie ligne 1 par effet de hoisting ➜ undefined
- Point 2 : la ligne 5 est exécutée
- Point 3 : la variable définie ligne 1 n'a jamais changé de valeur

ES6 : LET ET CONST

- `let` et `const` ont une portée de bloc et ne subissent pas le hoisting
- `let` : pour une variable normale (à la place de `var`)
- `const` : pour une variable dont on va interdire l'affectation

/var.js

```
1   var nom = "William Ruchaud";
2
3   function demo(cond) {
4       if(cond) {
5           var nom = "Hervé Gaudin";
6       }
7       return nom;
8   }
9
10  console.log("Point 1 : "+demo(false));
11  console.log("Point 2 : "+demo(true));
12  console.log("Point 3 : "+nom);
```

Portée de fonction

```
top  Filter
Point 1 : undefined
Point 2 : Herve Gaudin
Point 3 : William Ruchaud
>
```

/let.js

```
1   let nom = "William Ruchaud";
2
3   function demo(cond) {
4       if(cond) {
5           let nom = "Hervé Gaudin";
6       }
7       return nom;
8   }
9
10  console.log("Point 1 : "+demo(false));
11  console.log("Point 2 : "+demo(true));
12  console.log("Point 3 : "+nom);
```

Portée de bloc !

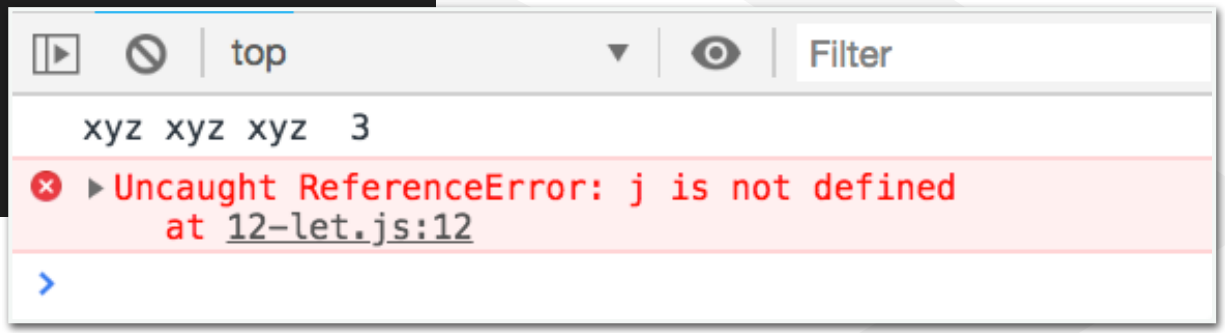
```
top  Filter
Point 1 : William Ruchaud
Point 2 : William Ruchaud
Point 3 : William Ruchaud
>
```

ES6 : LET ET CONST

- Avec `let` et `const`, la portée est celle du bloc de définition

/let-var.js

```
1   var t1 = "";
2   for(var i=0;i<3;i++) {
3       t1 += "xyz ";
4   }
5
6   var t2 = "";
7   for(let j=0;j<3;j++) {
8       t2 += "abc ";
9   }
10
11  console.log(t1+" "+i);
12  console.log(t2+" "+j);
```



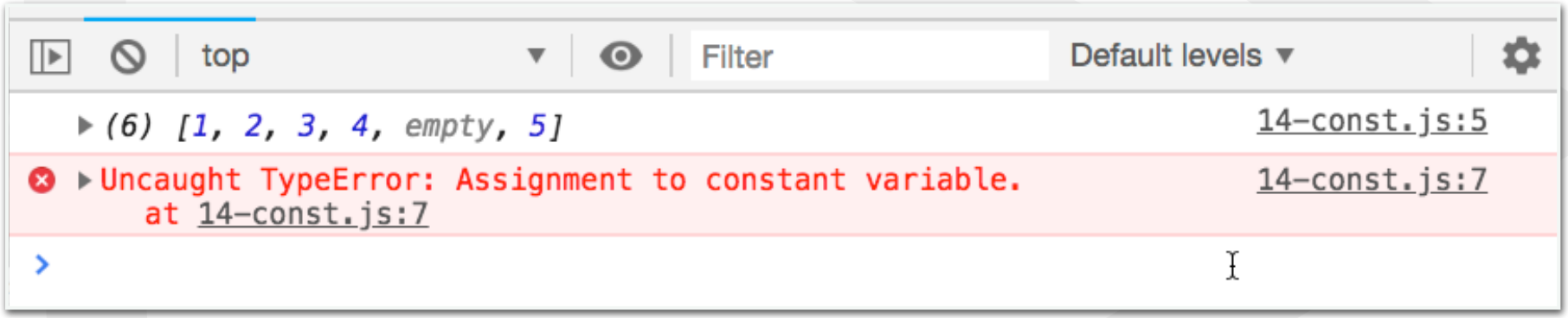
- Ligne 11 : `i` est disponible par effet de hoisting
- Ligne 12 : `j` n'est pas disponible car a sa portée limitée à la boucle `for()`

ES6 : LET ET CONST

- `const` permet d'interdire l'affectation à une variable
- Les opérations sur les tableaux et les objets restent possibles

/const.js

```
1   const tab = [1,2,3];
2   tab.push(4);
3   tab[5] = 5;
4
5   console.log(tab);
6
7   tab = [10,11,12];
```



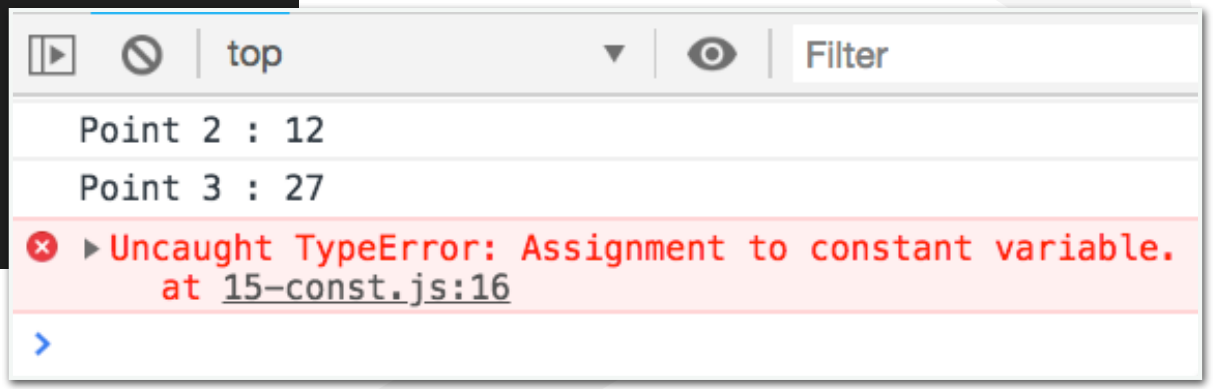
- Toutes les lignes passent sauf la 7 qui tente une affectation interdite

ES6 : LET ET CONST

- Idem avec les objets : l'accès y compris en affectation des propriétés est possible
- Seule la référence est protégée

/const.js

```
1   const obj = {
2       x : 6,
3       setX : function(x) {
4           this.x = x;
5       }
6   }
7
8   console.log("Point 1 : "+obj.x);
9
10  obj.x = 12;
11  console.log("Point 2 : "+obj.x);
12
13  obj.setX(27);
14  console.log("Point 3 : "+obj.x);
15
16  obj = {};
```



- La ligne 16 est la seule à enfreindre le `const`.

Rappels JavaScript
Partie 6

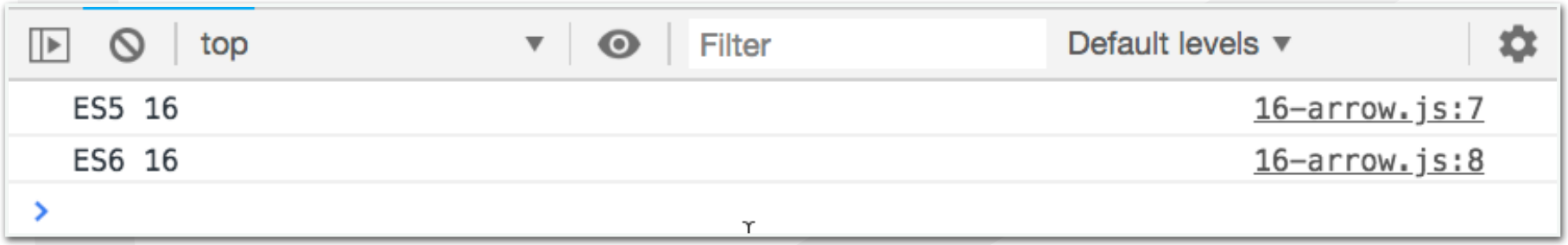
ES6 : FONCTIONS FLÈCHES

- Nouvelle façon d'écrire des fonctions en JavaScript
- Attention : ne pas confondre concision et efficacité

- Fonctions flèches => simplification de l'écriture

/fleche.js

```
1  var sommeES5 = function(x,y) {
2      return x+y;
3  }
4
5  let sommeES6 = (x,y) => x+y;
6
7  console.log("ES5 "+sommeES5(7,9));
8  console.log("ES6 "+sommeES6(7,9));
9
```



- Les deux fonctions réalisent des traitements équivalents

ES6 : FONCTIONS FLÈCHES

- Syntaxe modifiée suivant les cas

/fleche.js

```
1  let double  = x => 2*x;
2
3  let min = (x,y) => {
4      if(x<y) return x;
5      return y;
6  }
7
8  let creerPoint = (_x,_y) => ({x:_x,y:_y});
```

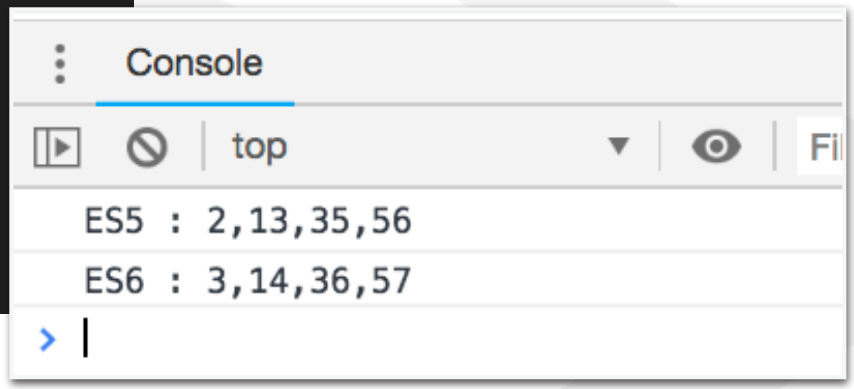
- Ligne 1 : un seul paramètre, les parenthèses sont inutiles
- Ligne 3 : code plus complexe, mettre les accolades
- Ligne 8 : retourne un objet, mettre les parenthèses autour du retour (erreur sinon).

ES6 : FONCTIONS FLÈCHES

- Exemple d'utilisation

/fleche.js

```
1   let tab1 = [1,12,34,55];
2
3   // façon ES5
4   tab1.forEach(function(val,index,t){
5       t[index]++;
6   });
7
8   console.log("ES5 : "+tab1);
9
10  // façon ES6
11  tab1.forEach((val,index,t) => t[index]++);
12
13  console.log("ES6 : "+tab1);
```



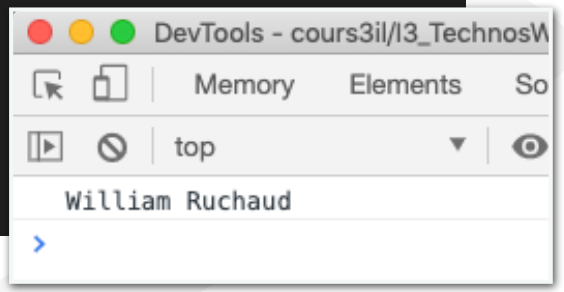
- Rappel : le callback de `Array.forEach()` est appelé avec pour paramètres :
  - La valeur courante
  - L'index de la valeur courante
  - Le tableau lui-même

ES6 : FONCTIONS ET CONTEXTE

- Notion de contexte : objet qui appèle la fonction (`this`)
  - En JavaScript toutes les fonctions et variables sont rattachées à un contexte qui peut être implicite (contexte global)
  - Ne pas confondre contexte (context en anglais) et la portée (scope en anglais)

/contexte.js

```
1  let prof = {
2      nom : "William Ruchaud",
3
4      afficher: function() {
5          console.log(this.nom);
6      }
7  }
8
9  prof.afficher();
```



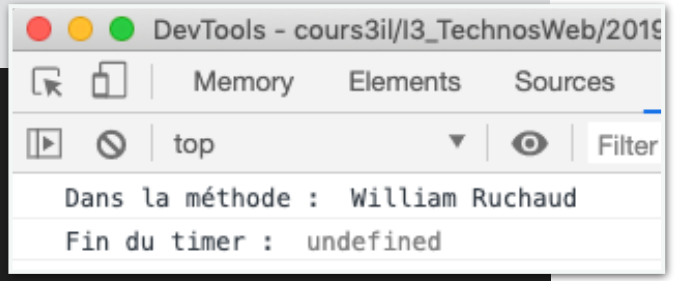
- Ici le contexte de `afficher()` est l'objet en train d'être créé

ES6 : FONCTIONS ET CONTEXTE

- Ajoutons un timer avec une fonction callback anonyme

/contexte.js

```
1   let prof = {
2       nom : "William Ruchaud",
3
4       afficher: function() {
5           console.log("Dans la méthode : ",this.nom);
6           setTimeout(function() {
7               console.log("Fin du timer : ",this.nom);
8           },2000)
9       }
10  }
11
12  prof.afficher();
```



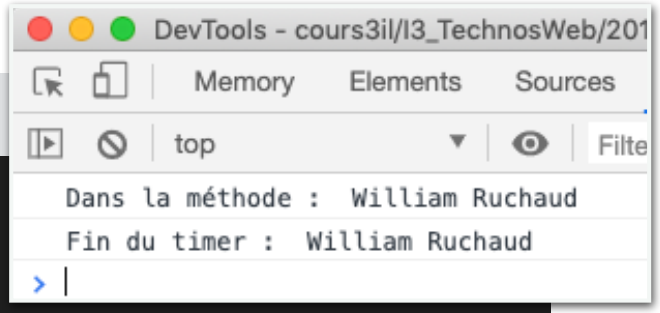
- À la fin du timer, `this.nom` retourne `undefined`.
- La raison est que la fonction callback n'est pas rattachée (en anglais bind / bound) au contexte de l'objet `prof`, mais au contexte global.
- De façon évidente, ce n'est pas une méthode de l'objet `prof`.

ES6 : FONCTIONS FLÈCHES ET CONTEXTE

- Voyons avec une fonction flèche à la place

/contexte.js

```
1   let prof = {
2       nom : "William Ruchaud",
3
4       afficher: function() {
5           console.log("Dans la méthode : ",this.nom);
6           setTimeout(() => {
7               console.log(" Fin du timer : ",this.nom);
8           },2000)
9       }
10  }
11
12  prof.afficher();
```



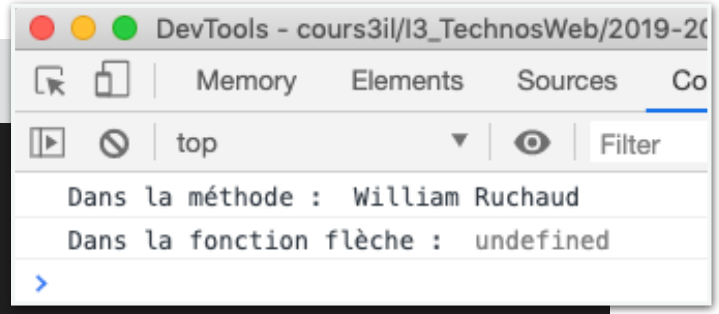
- Ici la fonction flèche est rattachée au contexte de l'objet `prof`.
- Fonctions flèches recommandées dans l'écriture des callback.

ES6 : FONCTIONS FLÈCHES ET MÉTHODES

- Utilisation d'une fonction flèche pour méthode

/contexte.js

```
1   let prof = {
2       nom : "William Ruchaud",
3
4       afficher: function() {
5           console.log("Dans la méthode : ",this.nom);
6       },
7
8       ecrire: () => {
9           console.log("Dans la fonction flèche : ",this.nom);
10      }
11
12  }
13
14  prof.afficher();
15  prof.ecrire();
```



- La méthode `ecrire()` ne connaît pas `this.nom`.
- Une fonction flèche récupère le `this` du contexte où elle est définie mais pas l'objet en cours de création.
- Fonctions classiques et fonctions flèches ne sont pas interchangeables

ES6 : FONCTIONS FLÈCHES VS FONCTIONS

> Ce tableau synthétise les principales différences

Caractéristique	Fonction classique	Fonction flèche
Utilisable en constructeur	OUI	NON
Capture du contexte (`this`)	NON	OUI
Accès aux arguments (`arguments`)	OUI	NON
Accès au prototype (`prototype`)	OUI	NON
Peut recevoir un nouveau contexte (`bind`, `apply`, `call`)	OUI	NON
Usage recommandé	Méthodes	Callbacks Fonctions sans usage du contexte

Rappels JavaScript
Partie 7

ES6 : CLASSES

- Rappel :
  - ES5 utilise une fonction comme constructeur
  - Notion de prototype ≠ notion de classe

- ES6 : possibilité de définir une classe,
  - Proche des modèles "classiques" type Java/PHP
  - Mais reste un sucre syntaxique pour la création d'un prototype

- Toujours pas de `public` / `protected` / `private`
- Héritage avec `extends`

ES6 : CLASSES

- Exemple

/classe-es5.js

```
1   function PointES5(x,y) {
2       this.x = x;
3       this.y = y;
4   }
5
6   PointES5.prototype.move = function (dx,dy) {
7       this.x += dx;
8       this.y += dy;
9   }
10
11  var p = new PointES5(3,6);
12  console.log(p.x);
```

/classe-es6.js

```
1   class PointES6 {
2       constructor(x,y) {
3           this.x = x;
4           this.y = y;
5       }
6
7       move(dx,dy) {
8           this.x += dx;
9           this.y += dy;
10      }
11  }
12
13  let p = new PointES6(5,7);
14  console.log(p.x);
```

- Rappel : en JavaScript les objets sont "très dynamiques"
  - Le prototype / la classe ne sont qu'un point de départ
  - Possibilité de champs propres ou de méthodes propres non définies dans le prototype / la classe

ES6 : CLASSES ET GETTER / SETTER

- Exemple :

/getter-setter.js

```
1   class Prof {
2       constructor(nom,bureau) {
3           this.nom = nom;          // Utilise le setter
4           this._bureau = bureau;
5       }
6
7       get nom() {
8           return this._nom;
9       }
10
11      set nom(nom) {
12          console.log("set nom");
13          this._nom = nom;
14      }
15
16      get bureau() {
17          console.log("get bureau");
18          return this._bureau;
19      }
20  }
21
22  var p = new Prof("William Ruchaud",210);
23  p.nom = "Hervé Gaudin";
24  p._bureau = 217;
25  console.log(p);
26  console.log(p.bureau);
```

- Attention aux noms !
- Nécessité d'avoir un nom de propriété différent du getter/setter
- Ici `_nom` et `_bureau`

```
set nom
set nom
▶ Prof {_nom: "Herve Gaudin", _bureau: 217}
get bureau
217
```

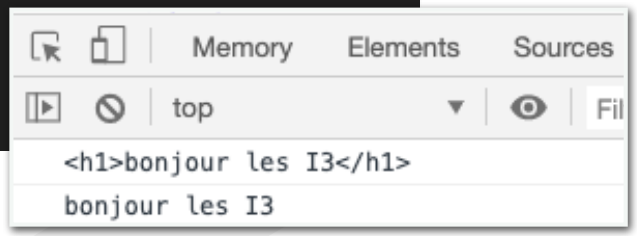
ES6 : CLASSES ET HÉRITAGE

Pour l'héritage, il faut utiliser le classique `extends`

/heritage.js

```
1   class Message {
2       constructor(txt) {
3           this.text = txt;
4       }
5
6       display() {
7           console.log(this.text);
8       }
9   }
10
11  class Tag extends Message {
12      constructor(balise,txt) {
13          super(txt);
14          this.balise = balise;
15      }
16
17      toHTML() {
18          console.log("<"+this.balise+">"+this.text+"</"+this.balise+">");
19      }
20  }
21
22  let h1 = new Tag("h1","bonjour les I3");
23  h1.toHTML();
24  h1.display();
```

Appel du constructeur parent



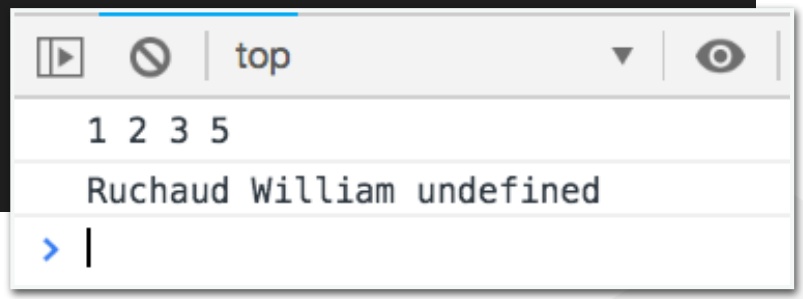
Rappels JavaScript
Partie 8

ES6 : DESTRUCTURATION

- Principe : faciliter l'extraction de données à partir d'un tableau ou d'un objet.

/destructuration.js

```
1  let [a,b,c,,d] = [1,2,3,4,5,6,7,8,9,10];
2
3  console.log(a,b,c,d);
4
5  let { prenom, nom, bureau } = {
6      nom : "Ruchaud",
7      prenom : "William",
8      fonction : "Directeur des études"
9  };
10
11 console.log(nom,prenom,bureau);
```



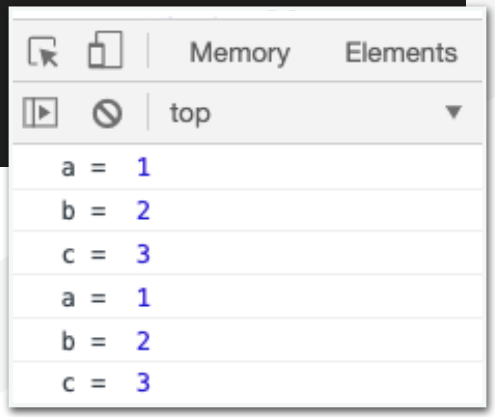
- Ligne 1 : les variables a,b,c et d sont initialisées à partir des valeurs situées aux mêmes emplacements dans le tableau (5e position pour d)
- Ligne 5 : les variables `prenom`, `nom` et `bureau` sont initialisées à partir de champs de même nom, s'ils existent dans l'objet, `undefined` sinon.

ES6 : OPÉRATEUR SPREAD (...)

- Spread operator : ...
- Sert à déstructurer les tableaux ou les objets

/spread.js

```
1   function demo(a,b,c) {
2       console.log("a = ",a);
3       console.log("b = ",b);
4       console.log("c = ",c);
5   }
6
7   let tab = [1,2,3,4,5,6];
8
9   // Appel façon "traditionnelle"
10  demo(tab[0],tab[1],tab[2]);
11
12  // Appel avec opérateur spread
13  demo(...tab);
```

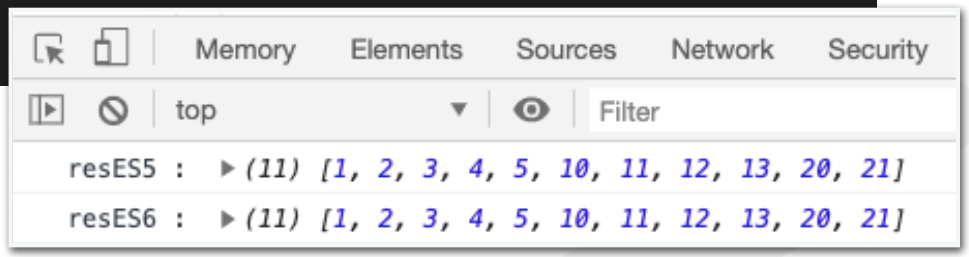


ES6 : OPÉRATEUR SPREAD (...)

- Facilite la concaténation de tableaux

/spread.js

```
1   let tab1 = [1,2,3,4,5];
2   let tab2 = [10,11,12,13];
3
4   // concaténation façon es5
5   let resES5 = tab1.concat(tab2,[20,21]);
6
7   // concaténation façon es6
8   let resES6 = [...tab1,...tab2,20,21];
9
10  console.log("resES5 : ",resES5);
11  console.log("resES6 : ",resES6);
```

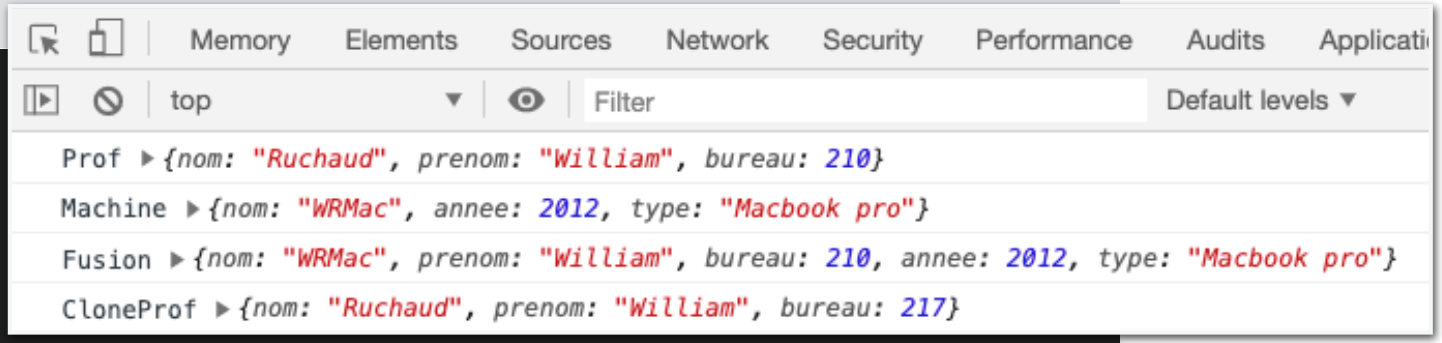


ES6 : OPÉRATEUR SPREAD (...)

- Facilite le clonage ou la fusion d'objets

/spread.js

```
1   let prof = {
2       nom : "Ruchaud",
3       prenom :"William",
4       bureau : 210
5   };
6
7   let machine = {
8       nom : "WRMac",
9       annee : 2012,
10      type : "Macbook pro"
11  };
12
13  let fusion = {...prof,...machine};
14  let cloneProf = {...prof};
15
16  cloneProf.bureau = 217;
17
18  console.log("Prof",prof);
19  console.log("Machine",machine);
20  console.log("Fusion",fusion);
21  console.log("CloneProf",cloneProf);
```

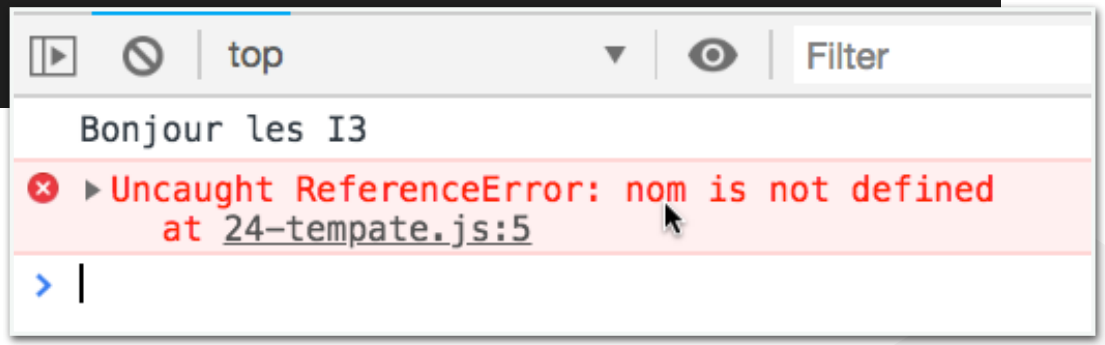


ES6 : TEMPLATE

- Pour faciliter les concaténations

/template.js

```
1  let groupe = "I3";
2  let template = `Bonjour les ${groupe}`;
3  console.log(template);
4
5  let message = `Enseignant ${nom} ${prenom}`;
6  let nom = "Ruchaud";
7  let prenom = "William";
8  console.log(message);
```



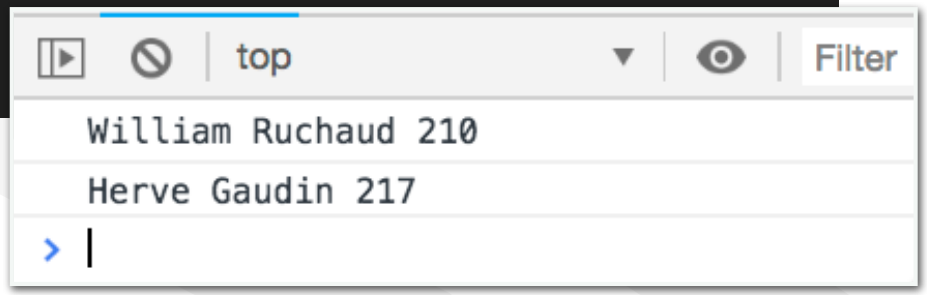
- Attention il faut utiliser ` pour baliser la chaîne (et non ' ou "), s'obtient avec `AltGr + 7` puis `espace` sur Windows
- Les variables utilisées dans le template doivent être définie avant l'usage.

ES6 : VALEUR PAR DÉFAUT

- Le paramètres peuvent avoir une valeur par défaut

/template.js

```
1  function afficherProf(nom,bureau=210) {
2      console.log(`${nom} ${bureau}`);
3  }
4
5  afficherProf("William Ruchaud");
6  afficherProf("Hervé Gaudin", 217)
```



- Rappel : JavaScript ne fait aucun contrôle des paramètres d'une fonction (type ou quantité)

ES6 : MODULES

- ES5 : quasi impossible d'inclure du JavaScript depuis du JavaScript (au sens `import` en Java).
  - Existence de rustines (`require.js`)
  - Nécessité de créer des balises `<script>` dans le fichier HTML

- ES6 : possibilité de créer des modules
  - Le fichier JavaScript "module" doit préciser ce qu'il exporte (fonction, classe,...)
  - Le fichier important module peut soit importer "à la carte" soit globalement le module

- **Important** :
  - Au moins un fichier doit être déclaré dans le fichier HTML
  - Utiliser `<script type="module" src="…">` au lieu de `<script src="…">`

ES6 : MODULES

Exemple

/index.html

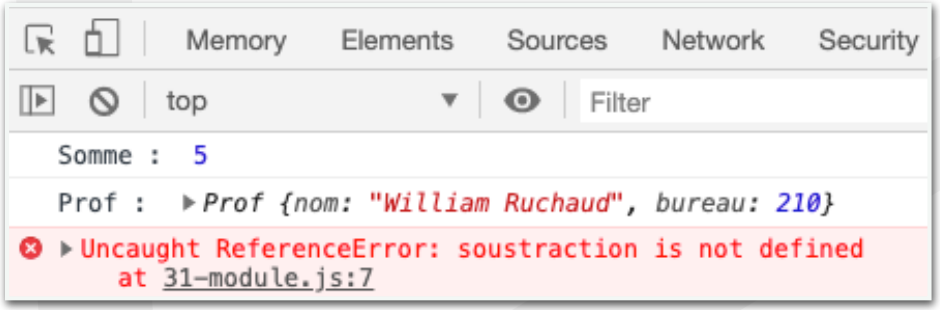
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <title>JavaScript Exemple</title>
</head>
<body>
    <script type="module" src="31-module.js"></script>
</body>
</html>
```

/31-module.js

```js
import { somme,Prof } from "./module3il.js";

let prof = new Prof("William Ruchaud",210);

console.log("Somme : ",somme(2,3));
console.log("Prof : ",prof);
console.log("Soustraction : ",soustraction(8,3));
console.log("Multiplication : ",multiplication(2,3));
```



/module3il.js

```js
function somme(x,y) {
    return x+y;
}

function soustraction(x,y) {
    return x-y;
}

function multiplication(x,y) {
    return x*y;
}

class Prof  {
    constructor(nom, bureau) {
        this.nom = nom;
        this.bureau = bureau;
    }
}

export {
    somme,soustraction,Prof
}
```

Rappels JavaScript
Partie 9

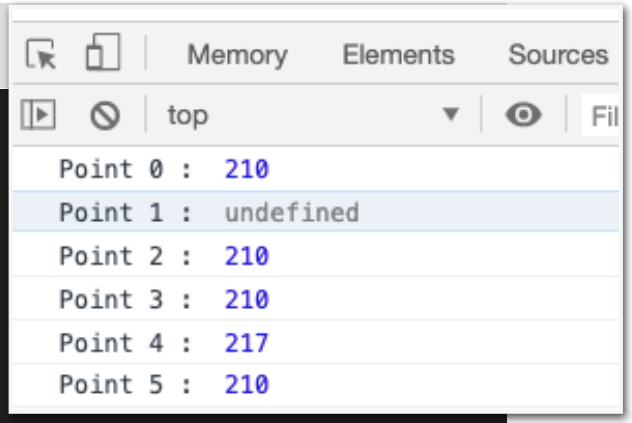
BIND (ES5)

- `bind` permet de construire une **nouvelle fonction** en lui associant un contexte d'exécution (`this`)

/32-bind.js

```
1   var profWR = {
2       nom:"William Ruchaud",
3       bureau:210,
4       getBureau:function() {
5           return this.bureau;
6       }
7   }
8
9   console.log("Point 0 : ",profWR.getBureau());
10
11  var alias = profWR.getBureau;
12  console.log("Point 1 : ",alias());
13
14  var bureauWR = alias.bind(profWR);
15  console.log("Point 2 : ",bureauWR());
16
17  var profHG = {
18      bureau:217
19  }
20
21  var bureauHG = alias.bind(profHG);
22  console.log("Point 3 : ",bureauWR());
23  console.log("Point 4 : ",bureauHG());
24  console.log("Point 5 : ",profWR.getBureau());
```

Retourne undefined car n'a pas de contexte pour this



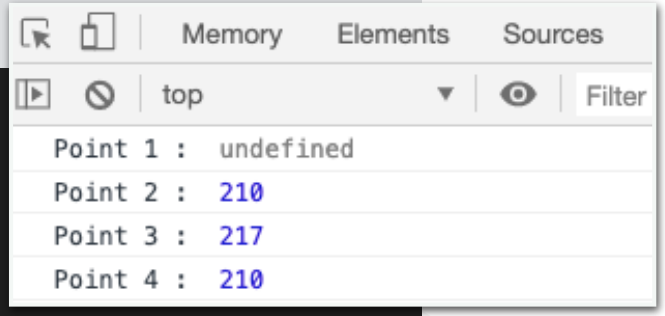
BIND (ES5)

- Possibilité d'ajout d'une méthode à partir d'une fonction obtenue sans contexte

/33-bind.js

```
1   var profWR = {
2       nom:"William Ruchaud",
3       bureau:210
4   }
5
6   var profHG = {
7       bureau:217
8   }
9
10  // Fonction définie en dehors de tout contexte
11  function getBureau() {
12      return this.bureau;
13  }
14
15  console.log("Point 1 : ",getBureau());
16
17  var getBureauWR = getBureau.bind(profWR);
18  console.log("Point 2 : ",getBureauWR());
19  console.log("Point 3 : ",getBureau.bind(profHG)());
20
21  profWR.monBureau = getBureau.bind(profWR);
22  console.log("Point 4 : ",profWR.monBureau());
```

Retourne undefined car n'a pas de contexte pour this

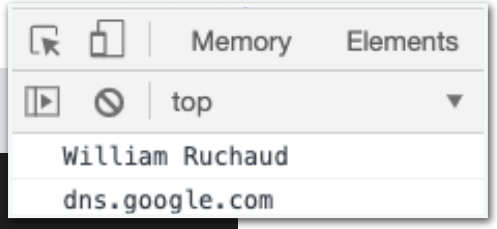


BIND (ES5)

- Possibilité de récupération d'une méthode à partir d'un objet existant (mais pas de la classe seule

/34-bind.js

```
1   class Prof {
2       constructor(nom,bureau) {
3           this.nom = nom;
4           this.bureau = bureau;
5       }
6
7       getNom() {
8           return this.nom;
9       }
10  }
11
12  class Machine {
13      constructor(nom,ip) {
14          this.nom = nom;
15          this.ip = ip;
16          this.getNom = (new Prof).getNom.bind(this);
17      }
18  }
19
20  let wr = new Prof('William Ruchaud',210);
21  console.log(wr.getNom());
22
23  let dns = new Machine("dns.google.com","8.8.8.8");
24  console.log(dns.getNom());
```



Récupère la méthode getNom()
à partir d'un objet