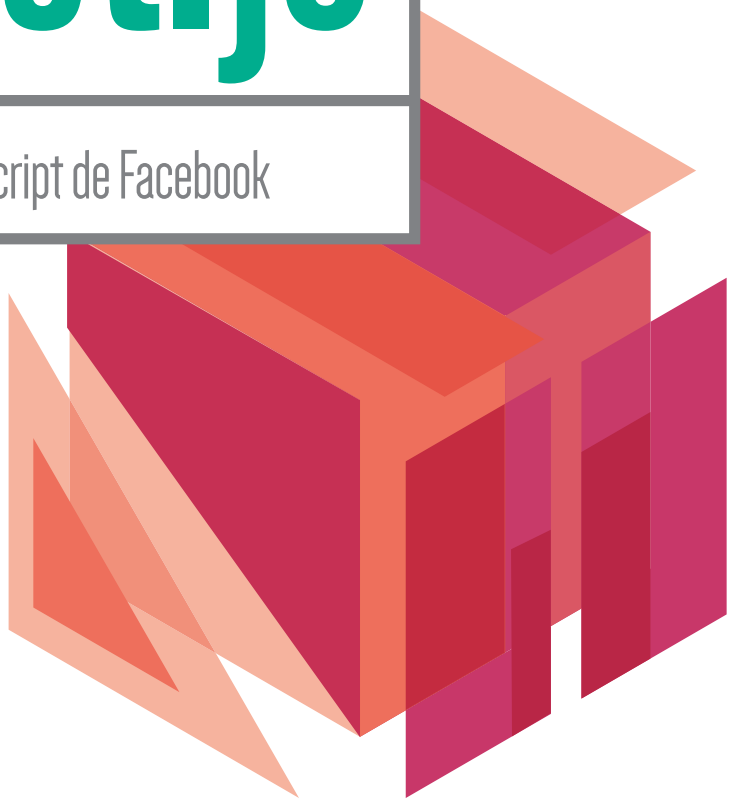
Éric Sarrion

React.js

LE framework JavaScript de Facebook



Éditions EYROLLES

React.js

Un ouvrage de référence pour les développeurs web

En tant que développeur, qui n'a pas encore entendu parler de React.js (ou React de façon raccourcie) ? Cette bibliothèque JavaScript, initialement écrite pour Facebook (en 2013), s'utilise maintenant couramment dans le monde de l'entreprise. Elle permet de structurer efficacement une application web, mais peut également s'utiliser dans une version dite native, pour écrire des applications mobiles à destination des iPhone ou Android.

Cet ouvrage vous permettra entre autres de créer des applications web autonomes, mais également de les interfacer avec un serveur en utilisant Ajax. Et surtout, vous comprendrez comment augmenter la complexité de votre application tout en conservant un code bien structuré, ceci grâce à React mais aussi Redux, étudié dans les derniers chapitres de l'ouvrage.

Agrémenté de nombreuses illustrations et de cas pratiques, cet ouvrage vous accompagne de façon progressive dans la découverte des concepts et propriétés associés à ce nouveau framework.

À qui s'adresse cet ouvrage ?

- Aux étudiants, développeurs et chefs de projet
- À tous les autodidactes férus de programmation qui veulent découvrir React.js

Au sommaire

Formateur et développeur en tant que consultant indépendant, **Éric Sarrion** participe à toutes sortes de projets informatiques depuis plus de 30 ans. Auteur des best-sellers *jQuery & jQuery UI*, *Programmation avec Node.js*, *Express.js et MongoDB*, et *jQuery mobile* aux éditions Eyrolles, il est réputé pour la limpidité de ses explications et de ses exemples.

www.editions-eyrolles.com

React.js

DANS LA MÊME COLLECTION

R. GOETTER. – **CSS 3 Grid Layout.**
N°67683, 2019, 131 pages.

C. BLAESS. – **Solutions temps réel sous Linux.**
N°67711, 3e édition, 2019, 318 pages.

T. PARISOT. – **Node.js.**
N°13993, 2018, 472 pages.

C. PIERRE DE GEYER, J. PAULI, P. MARTIN, E. DASPET. – **PHP 7 avancé.**
N°67720, 2e édition, 2018, 736 pages.

H. WICKHAM, G. GROLEMUND. – **R pour les data sciences.**
N°67571, 2018, 496 pages.

F. PROVOST, T. FAWCETT. – **Data science pour l'entreprise.**
N°67570, 2018, 370 pages.

J. CHOKOGOUE. – **Maîtrisez l'utilisation des technologies Hadoop.**
N°67478, 2018, 432 pages.

H. BEN REBAH, B. MARIAT. – **API HTML 5 : maîtrisez le Web moderne !**
N°67554, 2018, 294 pages.

W. MCKINNEY. – **Analyse de données en Python.**
N°14109, 2015, 488 pages.

E. BIERNAT, M. LUTZ. – **Data science : fondamentaux et études de cas.**
N°14243, 2015, 312 pages.

Éric Sarrion

React.js

Éditions
EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Table des matières

CHAPITRE 5
Interactions dans les composants React 119

CHAPITRE 6
Cas pratique : gérer les éléments d'une liste 133

CHAPITRE 7
Gérer les formulaires avec React 171

Avant-propos

Pourquoi un livre sur React.js ?

En tant que développeur, qui n'a pas encore entendu parler de React.js (ou React de façon raccourcie) ? Cette bibliothèque JavaScript, initialement écrite pour Facebook (en 2013), s'utilise maintenant couramment dans le monde de l'entreprise. Elle permet de structurer efficacement une application web, mais peut également s'utiliser dans une version dite native, pour écrire des applications mobiles à destination des iPhone ou Android (seule la version web est ici explorée).

Guide de lecture

Le livre est écrit de façon progressive pour le lecteur. Chaque chapitre propose des connaissances qui seront utilisées dans les chapitres suivants. Il est donc déconseillé de sauter un chapitre.

Le but final est de comprendre, pas à pas, comment fonctionne React afin de l'utiliser de façon professionnelle. Pour cela, le livre est organisé en 12 chapitres :

- Chapitre 1 - JavaScript ES6. Découvrir la nouvelle syntaxe de JavaScript utilisée par React.
- Chapitre 2 - Hello React. Introduction à React pour écrire le premier programme.
- Chapitre 3 - React et JSX. Écrire plus facilement le code React à l'aide de la syntaxe JSX.
- Chapitre 4 - Objet state. Comprendre le fonctionnement des états dans les composants React.
- Chapitre 5 - Interactions dans les composants React. Gérer les événements extérieurs, comme par exemple les clics sur les boutons.
- Chapitre 6 - Cas pratique : gérer les éléments d'une liste. Exemple complet pour interagir sur les éléments d'une liste (ajout, modification et suppression).
- Chapitre 7 - Gérer les formulaires avec React. Utiliser chaque élément de formulaire pour les utilisations les plus courantes avec React.

- Chapitre 8 - Utiliser create-react-app pour créer une application React. Utiliser un logiciel spécifique qui crée l'architecture de l'application React.
- Chapitre 9 - Redux. Utiliser la bibliothèque Redux pour gérer les états de l'application.
- Chapitre 10 - React et Redux. Apprendre à associer React et Redux.
- Chapitre 11 - Utiliser le module react-redux. Utiliser ce module pour faciliter son association avec React.
- Chapitre 12 - React Router : utiliser un gestionnaire de routes dans React.

Public concerné

Développeurs, étudiants, et chefs de projets seront intéressés par la lecture de cet ouvrage.

Remerciements

Nous remercions vivement l'équipe des éditions Eyrolles, ainsi que Gabriel Bieules, Eliza Gapenne et Jean François Bichet pour leur relecture attentive.

Et merci à vous, cher lecteur, de lire ce livre ! Si celui-ci vous apporte une meilleure compréhension du sujet, je vous serai reconnaissant de vos remarques positives que vous pourrez déposer sur les réseaux sociaux et sites d'achat en ligne !

1

JavaScript ES6

JavaScript, langage permettant des interactions faciles entre une page web et l'utilisateur, a connu diverses évolutions ces dernières années, dont la plus significative est celle de la version ES6 (abréviation de « ECMAScript 6 »). Cette nouvelle version concerne particulièrement :

- les variables : déclaration, portée et mise en forme dans les chaînes de caractères ;
- les fonctions : paramètres par défaut, nouvelle forme de déclaration des fonctions ;
- les objets et les tableaux : déstructuration et opérateur... ;
- les classes d'objets : création et dérivation ;
- les promesses : utilisation du processus asynchrone ;
- les modules : pour mieux structurer le code JavaScript.

React utilise de façon intensive ces nouveaux éléments. Ils seront détaillés dans les paragraphes suivants afin d'écrire et comprendre plus facilement le code React qui sera utilisé dans ce livre.

Quel éditeur de code utiliser ?

Vous pouvez utiliser n'importe quel éditeur de code, et si vous n'avez pas encore fait votre choix, nous vous conseillons d'utiliser Visual Studio Code (gratuit et maintenu par Microsoft).

Dans la suite du chapitre, nous utiliserons un fichier `index.html` qui contiendra le code JavaScript utilisé (au moyen de balises `<script>`). La structure de ce fichier est la suivante.

Fichier index.html

```
<html>

<head>
</head>

<body>
</body>

<script>

// Ici, le code JavaScript
// ...

</script>

</html>
```

Pour exécuter ce fichier HTML, vous disposez de deux méthodes :

- le sélectionner dans le gestionnaire de fichiers et le faire glisser dans la fenêtre d'un navigateur ;
- saisir l'URL http://localhost/react dans la barre d'adresses du navigateur, en supposant que vous avez au préalable lancé un serveur (PHP, Node.js, J2EE, etc.) et déposé le fichier `index.html` dans le répertoire `react` du serveur.

Dans ce chapitre, nous utilisons la première méthode.

Bien sûr, ce fichier ne contenant aucune ligne de code pour l'instant, son exécution produit une page blanche à l'écran.

Les variables

Divers mots-clés ont été ajoutés au langage JavaScript afin de modifier la portée des variables. Par ailleurs, le mot-clé `var` permettant de définir une variable locale est toujours actif, mais ses effets de bord ont été corrigés.

Utilisation de const

Le mot-clé `const` permet de définir une constante qui, par définition, ne pourra plus être modifiée. En cas de modification par le programme, une erreur JavaScript est provoquée.

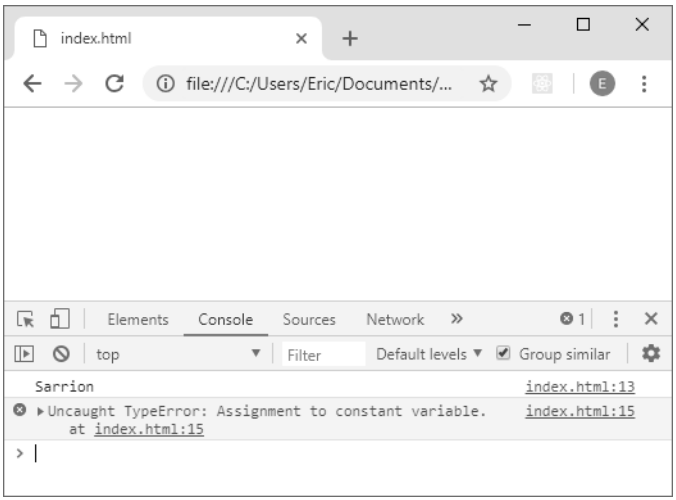
Écrivons le code suivant dans le fichier `index.html`, dans la partie réservée (balise `<script>`).

Utilisation de const

```
const nom = "Sarrion";
console.log(nom);
nom = "Martin";    // Erreur
```

La modification de la constante `nom` provoque une erreur que l'on peut voir dans un navigateur, par exemple Chrome (figure 1-1), en utilisant ses outils de développement (touche *F12* puis onglet *Console*).

Figure 1–1



Une constante (définie par `const`) ne peut plus être affectée une seconde fois. Seule la première affectation est autorisée, tandis que les suivantes produisent une erreur nous permettant de corriger le bug.

Utilisation de let

Le mot-clé `let` permet de définir de vraies variables locales, qui disparaissent lorsque le bloc de code dans lequel elles sont définies n'est plus exécuté. De plus, si une variable du même nom est définie à un niveau supérieur dans le code, cette nouvelle variable définie avec `let` n'écrase pas la valeur de la variable de niveau supérieur. Le comportement de `let` est bien différent de celui de `var`.

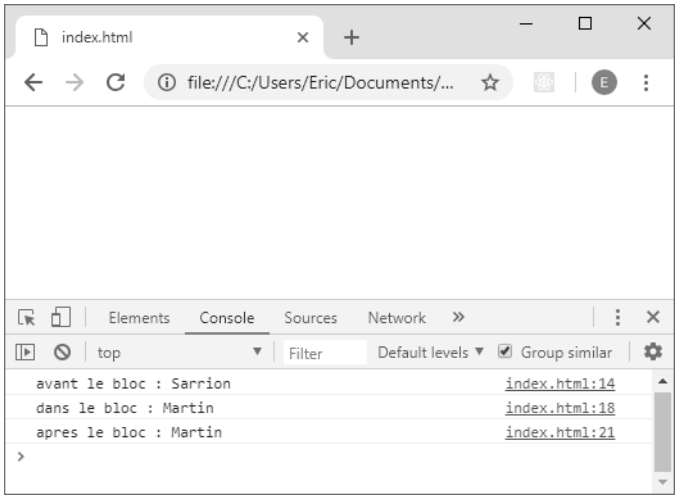
Voyons sur un exemple la différence entre l'utilisation de `var` et celle de `let`. La différence se voit lorsque le mot-clé `var` ou `let` est utilisé dans un bloc de code (entouré par des accolades).

Avec utilisation de var dans un bloc

```
var nom = "Sarrion";

console.log("avant le bloc : " + nom);    // "Sarrion"

if (true) {
  var nom = "Martin";
  console.log("dans le bloc : " + nom);   // "Martin"
}

console.log("apres le bloc : " + nom);    // "Martin"
```

La variable `nom` est définie en premier hors du bloc (à la valeur `"Sarrion"`). Une variable du même nom est créée dans le bloc, affectée avec une nouvelle valeur (`"Martin"`). Cette variable, déclarée à l'aide de `var`, vient écraser la variable du même nom définie avant le bloc. Cette variable modifiée est ensuite affichée après le bloc avec la nouvelle valeur.

Figure 1–2



La variable `nom` définie dans le bloc vient écraser la même variable (du même nom) définie avant le bloc, car la variable définie dans le bloc n'est pas vue comme étant locale au bloc. En fait, on n'a pas deux variables, mais une seule en mémoire.

L'effet d'écrasement (de la variable) observé vient du fait que la variable `nom` définie dans le bloc à l'aide de `var` n'est pas locale à ce bloc, mais vient prendre la même place qu'une éventuelle variable du même nom définie précédemment.

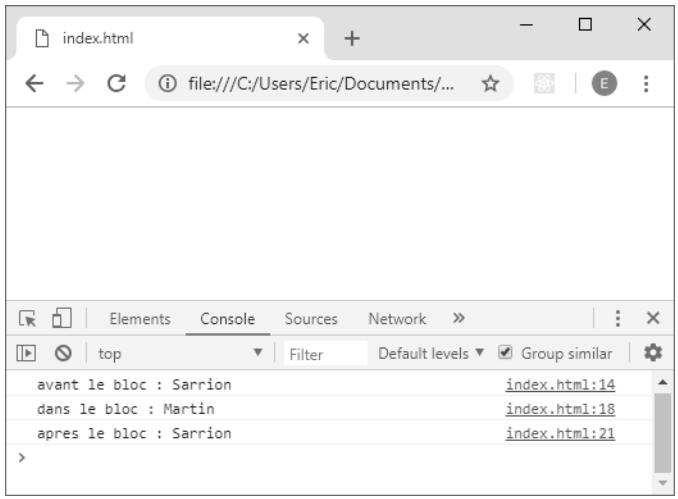
L'utilisation du mot-clé `let` va permettre de créer réellement des variables locales dans un bloc, sans interférer avec d'autres variables du même nom définies ailleurs.

Avec utilisation de let dans un bloc

```
var nom = "Sarrion";

console.log("avant le bloc : " + nom);    // "Sarrion"

if (true) {
  let nom = "Martin";
  console.log("dans le bloc : " + nom);   // "Martin"
}

console.log("apres le bloc : " + nom);    // "Sarrion"
```

On utilise maintenant le mot clé `let` pour définir la variable dans le bloc. Le résultat va être très différent du précédent exemple…

Figure 1–3



La variable `nom` a maintenant une valeur différente dans le bloc et en dehors de celui-ci. C'est l'utilisation de `let` (dans le bloc) qui le permet.

On peut également observer la différence entre `var` et `let` sur un second exemple. Dans une boucle `for()`, une variable déclarée par `var` ou par `let` a une incidence sur le comportement du code JavaScript.

Modifions le fichier `index.html` pour créer cinq éléments `<div>` sur lesquels on intercepte le clic sur chacun d'eux. On utilise tout d'abord le mot-clé `var` pour définir l'indice `i` dans la boucle.

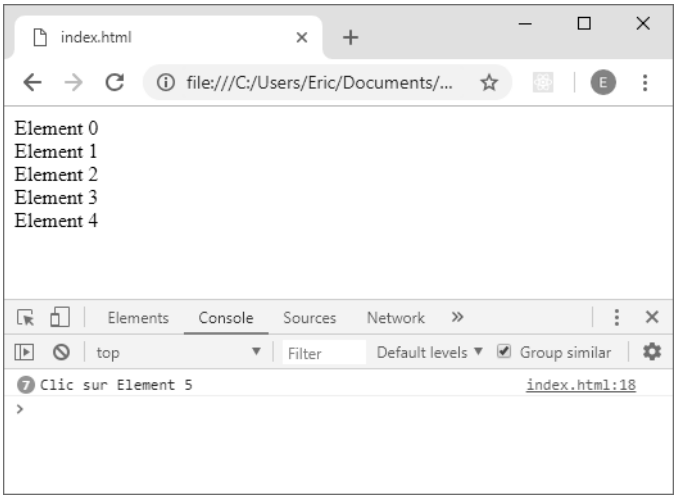
Clic sur les éléments créés avec un indice de boucle défini par var dans une boucle for()

```
for (var i=0; i<5; i++) {   // Utilisation de var pour définir l'indice i dans le bloc
  var div = document.createElement("div");
  var text = document.createTextNode("Element " + i);
  div.appendChild(text);
  document.getElementsByTagName("body")[0].appendChild(div);

  div.onclick = function() {
    console.log("Clic sur Element " + i);
  }
}
```

Ce programme effectue simplement une boucle de 0 à 4 inclus, afin de créer des éléments `<div>` sur lesquels on positionne un gestionnaire d'événement `onclick`. Chaque clic sur un élément `<div>` affiche `"Clic sur Element "` suivi de l'index de l'élément `<div>` sur lequel on a cliqué.

L'intérêt de ce petit programme ne réside pas dans le code JavaScript permettant de créer les éléments `<div>` dans la boucle, mais plutôt dans l'observation de la valeur affichée dans la console lors des clics sur les différents éléments `<div>` de la page. Chaque clic sur un élément produit l'affichage de « Clic sur Element 5 », sachant que cet élément 5 n'est même pas présent dans la page (figure 1-4) !

Figure 1–4



En effet, la variable `i` définie par `var` n'est pas locale au bloc dans lequel elle est définie, et vient donc écraser son ancienne valeur. A la fin de la boucle, elle finit par atteindre la valeur 5, ce qui provoque la fin de la boucle. Par la suite, le clic sur n'importe quel élément `<div>` récupère la valeur finale de cette variable, ici la valeur 5.

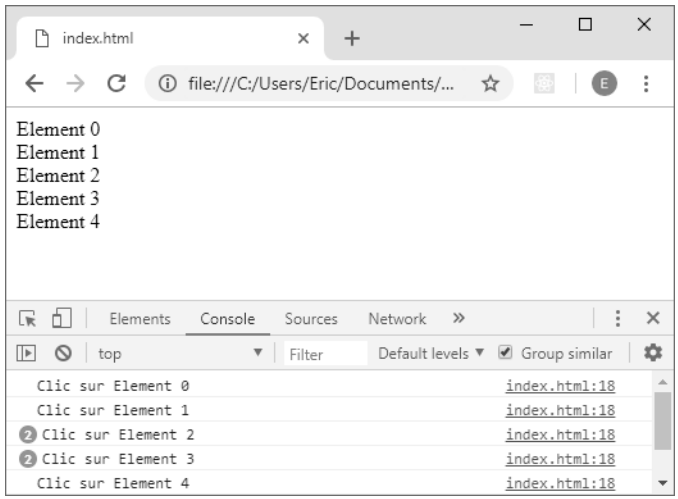
Un comportement bien différent est visible si l'on utilise le mot-clé `let` au lieu de `var` pour définir la variable `i` dans la boucle `for()`.

Clic sur les éléments créés avec un indice de boucle défini par let dans une boucle for()

```
for (let i=0; i<5; i++) {   // Utilisation de let pour définir i dans le bloc
  var div = document.createElement("div");
  var text = document.createTextNode("Element " + i);
  div.appendChild(text);
  document.getElementsByTagName("body")[0].appendChild(div);

  div.onclick = function() {
    console.log("Clic sur Element " + i);
  }
}
```

Le programme est identique au précédent, sauf que `let` a remplacé `var` pour définir la variable de boucle `i`.

Figure 1–5



On voit clairement que la variable `i` définie par `let` est maintenant locale à la boucle `for()`. Et chaque clic indique bien l'élément sur lequel on a cliqué.

Mise en forme des chaînes de caractères

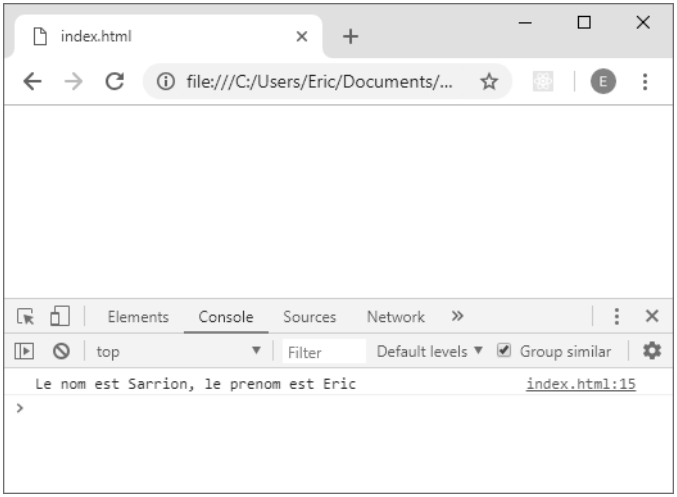
L'utilisation de variables dans une chaîne JavaScript est un peu fastidieuse car il faut fermer chaque chaîne de caractères avant d'utiliser la variable à concaténer à cette chaîne. On écrit souvent des lignes de code telles que celles-ci :

Concaténation de variables dans les chaînes de caractères

```
var nom = "Sarrion";
var prenom = "Eric";

var txt = "Le nom est " + nom + ", le prenom est " + prenom;
console.log(txt);
```

On voit que la ligne permettant de définir la variable `txt` oblige de fermer chaque chaîne de caractères avant d'utiliser une variable lors de la concaténation. Le résultat est celui attendu.

Figure 1–6



Une façon plus claire d'écrire peut être utilisée en ES6, sans fermer la chaîne de caractères lors de la concaténation des variables. On utilise pour cela le guillemet simple inversé (`) pour définir la chaîne de caractères (au lieu du simple (') ou double (") guillemet traditionnel). Le programme peut alors s'écrire :

Concaténation de variables dans les chaînes de caractères avec ES6

```
var nom = "Sarrion";
var prenom = "Eric";

var txt = `Le nom est ${nom}, le prenom est ${prenom}`;
console.log(txt);
```

Chaque variable est utilisée dans la chaîne au moyen de `${variable}`, sachant que la chaîne est entourée des guillemets simples inversés. Le résultat est identique au précédent, mais l'écriture est plus simple.

Le résultat est identique au précédent, et la forme d'écriture est plus concise.

Les fonctions en ES6

La déclaration des fonctions a été améliorée en ES6, pour la rendre plus performante et plus concise à écrire.

Utilisation de paramètres par défaut

On va maintenant pouvoir passer des valeurs par défaut à des paramètres de fonction, comme cela se fait dans d'autres langages de programmation. Pour cela, il suffit de définir les paramètres de la fonction en indiquant les valeurs par défaut qu'ils doivent avoir (pour ceux qui en ont), avec le signe =.

La règle est de définir les paramètres par défaut en fin de déclaration des paramètres, dans la fonction. Dès qu'une valeur par défaut est indiquée pour un paramètre, tous les autres paramètres qui suivent doivent également avoir une valeur par défaut définie (sinon une ambiguïté se crée lors de l'appel de la fonction, et il faut dans ce cas indiquer la valeur `undefined` pour cet argument lors de l'appel). C'est pourquoi ces paramètres sont indiqués par défaut à la fin de la déclaration de la fonction.

Utilisation des valeurs par défaut dans les fonctions

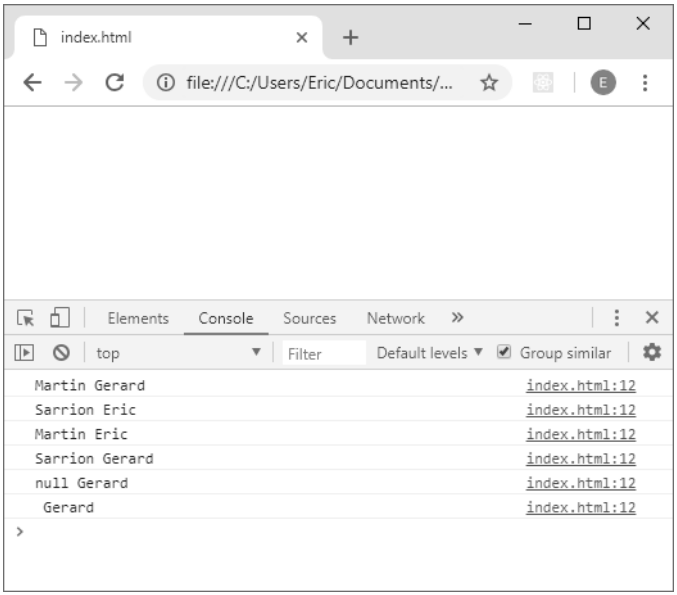
```
function log(nom="Sarrion", prenom="Eric") {
  console.log(`${nom} ${prenom}`);
}

log("Martin", "Gerard");    // "Martin Gerard"
log();                      // "Sarrion Eric"
log("Martin");              // "Martin Eric"
log(undefined, "Gerard");   // "Sarrion Gerard"
log(null, "Gerard");        // "null Gerard"
log("", "Gerard");          // "Gerard"
```

La fonction `log()` possède deux paramètres ayant des valeurs par défaut, et nous utilisons la fonction avec 0, 1 ou 2 arguments afin de voir son comportement (figure 1-7, page suivante).

On voit que lorsqu'un argument n'est pas utilisé, il est remplacé par sa valeur par défaut. Toutefois, comme tous les arguments sont ici facultatifs, il faut utiliser la valeur `undefined` lors de l'appel de la fonction si l'argument n'est pas le dernier (les valeurs `null` ou `""` ne sont pas remplacées par les valeurs par défaut).

Voici une variante de ce programme en mettant par défaut uniquement le premier paramètre (ce qui n'est pas conseillé car cela produit des ambiguïtés). Le comportement du programme est tout autre.

Figure 1–7



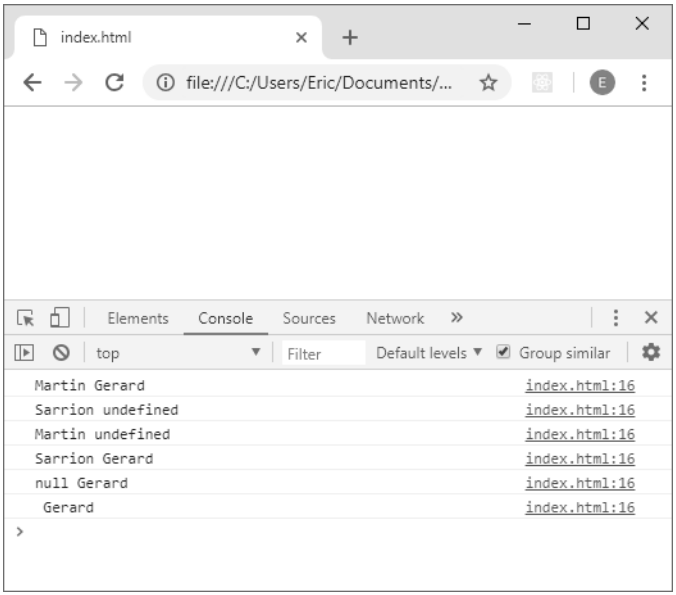
Utilisation d'une valeur par défaut dans le premier paramètre de la fonction

```
function log(nom="Sarrion", prenom) {
  console.log(`${nom} ${prenom}`);
}

log("Martin", "Gerard");    // "Martin Gerard"
log();                      // "Sarrion undefined"
log("Martin");              // "Martin undefined"
log(undefined, "Gerard");   // "Sarrion Gerard"
log(null, "Gerard");        // "null Gerard"
log("", "Gerard");          // "Gerard"
```

L'ambiguïté porte ici sur l'appel à `log("Martin")`, pour lequel l'argument `"Martin"` est considéré comme le premier paramètre de la fonction. Pour lever l'ambiguïté, on est obligé de mentionner `undefined` en premier argument lors de l'appel à `log(undefined, "Gerard")`, ce qui permet de remplacer l'argument `undefined` par sa valeur par défaut.

On retiendra donc que les paramètres par défaut s'ajoutent en priorité à la fin de la liste des paramètres lors de la définition des fonctions.

Figure 1–8



Nouvelle forme de déclaration des fonctions en ES6

Prenons la fonction `log(nom, prenom)` écrite précédemment. On peut également l'écrire sous la forme suivante :

Fonction log() écrite en ES6

```
var log = (nom="Sarrion", prenom="Eric") => {
  console.log(`${nom} ${prenom}`);
};
```

Le mot-clé `function` a disparu, remplacé par le signe `=>`. Les paramètres de la fonction s'écrivent toujours entre parenthèses, tandis que le corps de la fonction est toujours entouré des accolades de début et de fin.

Si la fonction n'a pas de paramètres, on l'écrit de la façon suivante.

Fonction log() écrite en ES6

```
var log = () => {
  console.log("Bonjour");
};
```

L'appel de la fonction s'effectue toujours de la même manière qu'auparavant (seule la définition peut s'effectuer de façon différente).

Appel à la fonction log()

```
log();            // Sans paramètres
log("Sarrion");   // Avec paramètres
```

Objet this dans les fonctions

Cette façon de définir les fonctions en ES6 peut avoir des incidences sur la valeur de l'objet `this`, qui représente l'objet en cours d'utilisation.

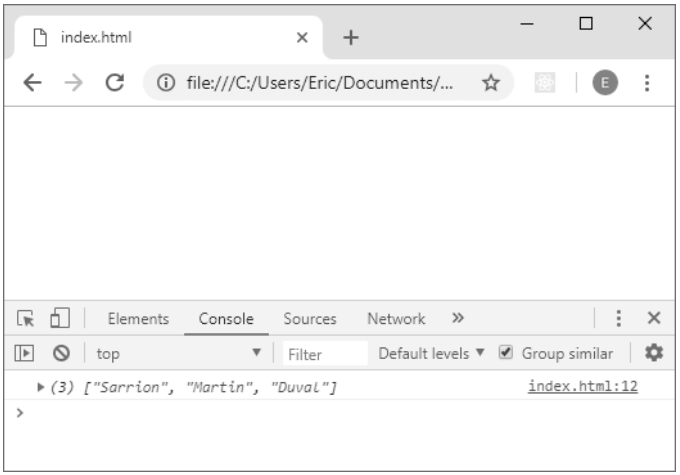
Considérons par exemple l'objet suivant, dans lequel est défini une liste de noms à afficher au moyen de la fonction `log()` définie également dans l'objet. On utilise l'ancienne forme de déclaration des fonctions (avec le mot-clé `function` au lieu de =>).

Objet avec fonction log() incorporée (non ES6)

```
var obj = {
  noms : ["Sarrion","Martin","Duval"],
  log: function() {
    console.log(this.noms);
  }
}

obj.log();   // ["Sarrion","Martin","Duval"]
```

L'objet `this` représente ici l'objet en cours d'utilisation de la fonction `log()`, donc l'objet `obj`.

Figure 1–9



L'objet `this` permet d'accéder aux noms, qui sont ainsi affichés dans la fonction `log()`.

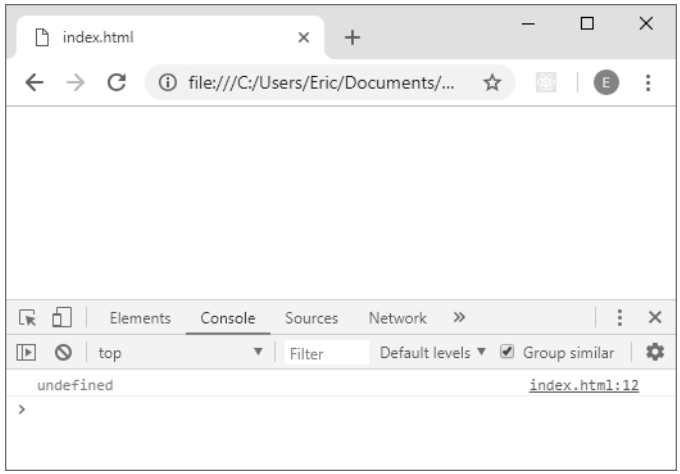
À présent, écrivons la fonction en tenant compte de la notation ES6.

Objet avec fonction log() incorporée (en ES6)

```
var obj = {
  noms : ["Sarrion","Martin","Duval"],
  log: () => {
    console.log(this.noms);
  }
}

obj.log();
```

Il s'agit du même programme écrit en notation ES6. Le résultat devrait être identique... Pourtant, lors de l'exécution de ce programme, on observe que le résultat est différent du précédent (figure 1-10) !

Figure 1–10

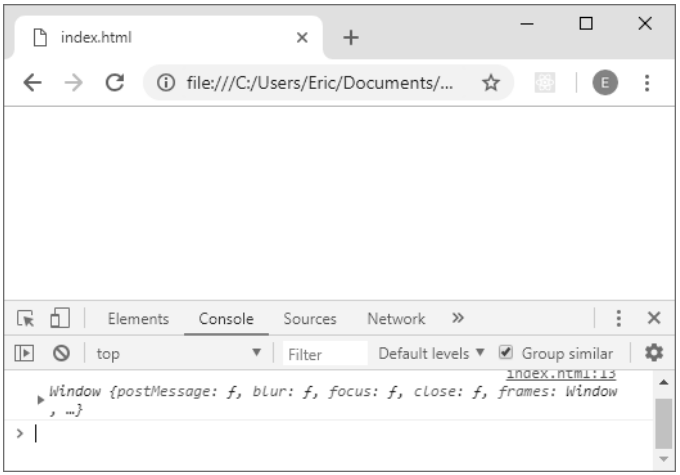


La propriété `noms` de l'objet n'est plus accessible (et vaut donc `undefined`) car `this` représente maintenant l'objet JavaScript `window`, et non plus l'objet `obj`. Pour s'en assurer, il suffit d'afficher la valeur de `this` dans la console.

Affichage de this dans la console

```
var obj = {
  noms : ["Sarrion","Martin","Duval"],
  log: () => {
    console.log(this);   // window
  }
}

obj.log();
```

Figure 1–11



C'est bien l'objet `window` qui est référencé par `this`.

Cette observation peut avoir un intérêt lors de l'utilisation de fonctions asynchrones, très utilisées avec JavaScript.

Considérons que l'on affiche les noms inscrits dans l'objet `obj` précédent, au bout de quelques millisecondes, en utilisant un timer JavaScript. Ce timer s'écrit au moyen d'une fonction asynchrone définie par `setTimeout()`.

Afficher les noms au bout de 10 ms

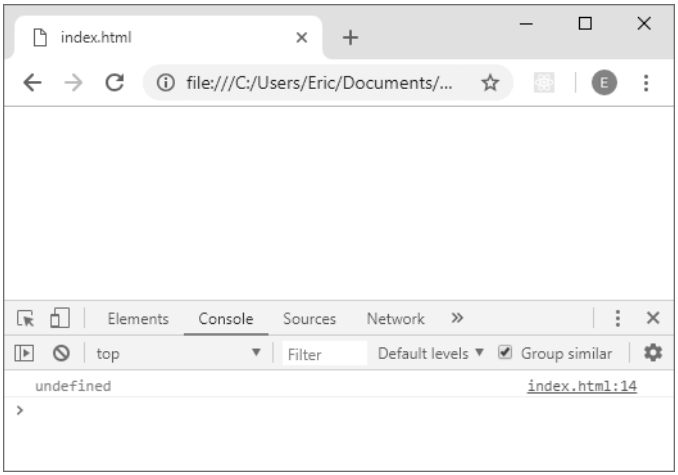
```
var obj = {
  noms : ["Sarrion","Martin","Duval"],
  log : function() {
    setTimeout(function() {
      console.log(this.noms);    // undefined
    }, 10);  // Exécution du timer au bout de 10 ms
  }
}

obj.log();
```

L'objet `this` est maintenant utilisé dans la fonction `setTimeout()`, elle-même définie sur l'objet `window` de la page HTML.

L'objet `this` représente donc l'objet `window` (c'est-à-dire l'objet qui utilise la fonction `setTimeout()`, comme si nous avions écrit dans le code `window.setTimeout()`).

L'utilisation d'une fonction asynchrone modifie complètement le comportement de notre programme. La notation ES6 permet de remédier à ce problème.

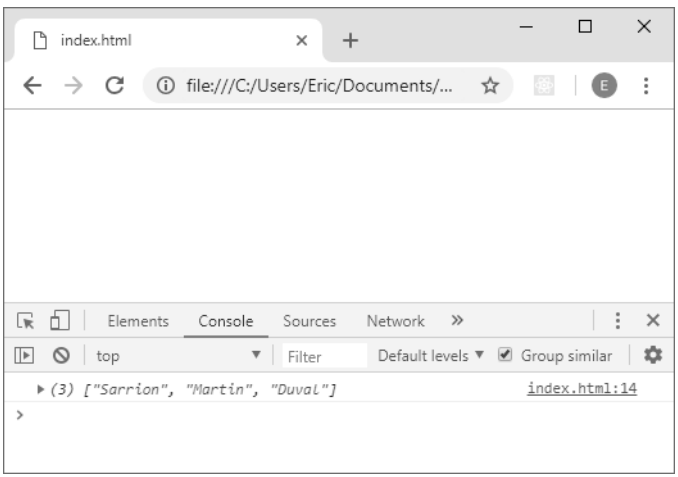
Figure 1–12



Utiliser la notation ES6 pour définir une fonction asynchrone

```
var obj = {
  noms : ["Sarrion","Martin","Duval"],
  log : function() {
    setTimeout(() => {          // Notation ES6
      console.log(this.noms);
    }, 10);
  }
}

obj.log();
```

Figure 1–13



Le résultat est maintenant conforme à nos attentes. La valeur `this` de l'objet est bien conservée et transmise dans la fonction asynchrone.

Les objets

Afin de manipuler plus aisément les objets, ES6 permet de les structurer/déstructurer, notions qui sont détaillées dans les paragraphes qui suivent.

Déstructurer un objet

Commençons par expliquer la déstructuration d'un objet. Elle permet d'accéder aisément aux propriétés d'un objet qui nous intéressent, sans nous préoccuper de celles qui ne nous intéressent pas dans l'immédiat.

Prenons l'exemple de l'objet `personne`, composé des attributs `nom`, `prenom` et `ville` affectés à cette personne.

Objet personne

```
var personne = {
  nom : "Sarrion",
  prenom : "Eric",
  ville : "Paris"
}
```

Afin d'accéder au `nom` et `prenom` de la personne, par exemple, on peut écrire en ES6 le code suivant.

Accès au nom et prenom de la personne

```
var { nom, prenom } = personne;
```

Cela crée deux variables `nom` et `prenom`, possédant respectivement la valeur de `personne.nom` et `personne.prenom`. On appelle cela la déstructuration d'un objet, car on accède uniquement aux propriétés intéressantes (dans cet endroit du programme).

Si seul le `nom` nous intéresse, on écrira la ligne suivante.

Accès uniquement au nom de la personne

```
var { nom } = personne;
```

Attention

Même si la notation permettant la déstructuration s'écrit avec des accolades, elle ne crée pas un nouvel objet, mais uniquement des variables qui correspondent aux valeurs des propriétés de l'objet auxquelles on accède. Ici on crée donc la variable `nom` dont la valeur est `personne.nom`.

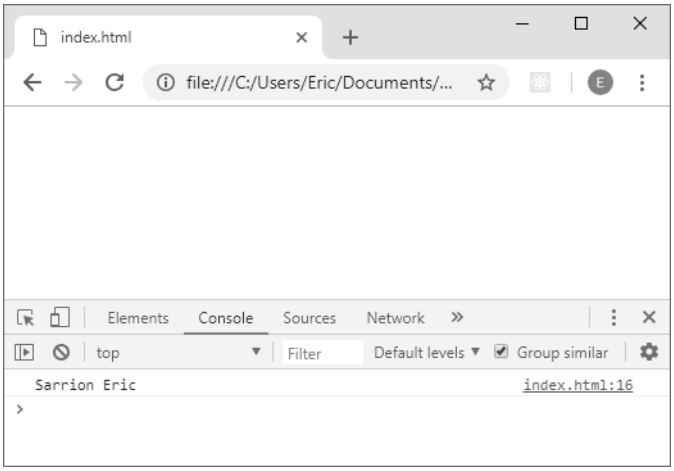
Cette notation sera particulièrement utile pour transmettre un objet en paramètre, en montrant clairement quelles sont les propriétés de l'objet qui sont finalement utilisées dans la fonction. Utilisons-la, par exemple, pour transmettre l'objet `personne` dans la fonction `log(personne)`, sachant que la fonction `log()` n'utilise que le `nom` et le `prenom` de la personne (on suppose ici que la propriété `ville` définie pour l'objet `personne` n'est pas utilisée dans la fonction `log()`).

Fonction `log()` définie en indiquant les propriétés réellement utilisées dans l'objet transmis en paramètres

```
var personne = {
  nom : "Sarrion",
  prenom : "Eric",
  ville : "Paris"
}

var log = ({nom, prenom}) => {    // Seules les propriétés nom et prenom seront
                                  // utilisées dans la fonction
  console.log(`${nom} ${prenom}`);
}
log(personne);  // La fonction est appelée avec l'objet en argument
```

On transmet un objet `personne` lors de l'appel de la fonction `log()`, mais seuls le `nom` et `prenom` sont utilisés dans celle-ci. Plutôt que d'indiquer un objet `personne` en paramètre de la fonction, on indique les réelles propriétés qui nous intéressent dans la fonction en les écrivant sous forme déstructurée `{ nom, prenom }` dans la liste des paramètres.

Figure 1–14



Le même programme, écrit de manière traditionnelle et donc sans utiliser la déstructuration des objets, peut s'écrire de la façon suivante.

Fonction log() écrite sans utiliser la déstructuration des objets

```
var personne = {
  nom : "Sarrion",
  prenom : "Eric",
  ville : "Paris"
}

var log = (p) => {
  console.log(`${p.nom} ${p.prenom}`);
}
log(personne);
```

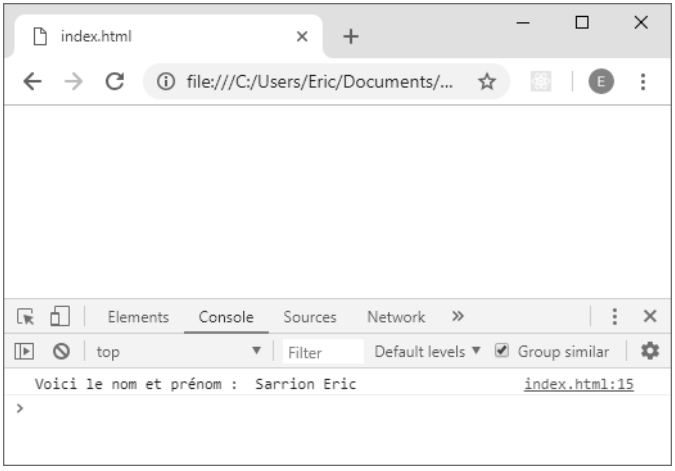
La fonction `log()` utilise ici le paramètre `p` correspondant à une personne. Rien n'indique, dans la liste des paramètres, que ce qui nous intéresse réellement dans le corps de la fonction est le `nom` et le `prenom`. Alors que l'utilisation de la forme déstructurée permet de bien voir les propriétés réellement utilisées dans l'objet passé en paramètres.

Si la fonction `log()` possède d'autres paramètres, il suffit de les indiquer dans la liste lors de sa définition. Supposons que l'on veuille indiquer en paramètre un texte qui sera affiché devant le `nom` et le `prenom`. On écrira alors le code qui suit.

Fonction log() avec plusieurs paramètres

```
var personne = {
  nom : "Sarrion",
  prenom : "Eric",
  ville : "Paris"
}

var log = ({nom, prenom}, texte) => {
  console.log(`${texte} ${nom} ${prenom}`);
}
log(personne, "Voici le nom et prénom : ");
```

Figure 1–15



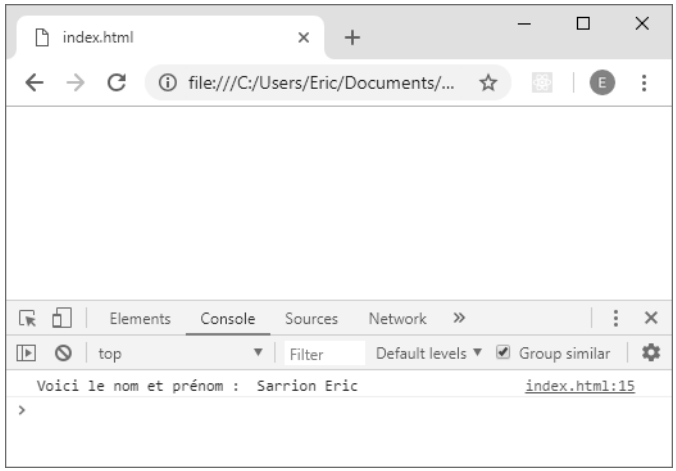
On peut également utiliser des valeurs par défaut pour les paramètres de la fonction. Par exemple, indiquons que le texte affiché a une valeur par défaut (utilisée si le texte n'est pas indiqué lors de l'appel de la fonction), et que le nom par défaut est `"Sarrion"` s'il n'est pas indiqué dans les propriétés de l'objet `personne` transmis lors de l'appel.

Utilisation de valeurs par défaut dans les paramètres de la fonction log()

```
var personne = {    // Objet personne défini sans le nom
  prenom : "Eric",
  ville : "Paris"
}

var log = ({nom="Sarrion", prenom}, texte="Voici le nom et prénom : ") => {
  console.log(`${texte} ${nom} ${prenom}`);
}
log(personne);     // Un seul argument lors de l'appel
```

L'objet `personne` est défini sans la propriété `nom`, qui aura une valeur par défaut lors de la définition de la fonction `log()`.

Figure 1–16



Structurer un objet

La structuration d'un objet est le processus inverse de la déstructuration vue précédemment. Elle permet de créer un objet JavaScript à partir de variables définies dans le code JavaScript.

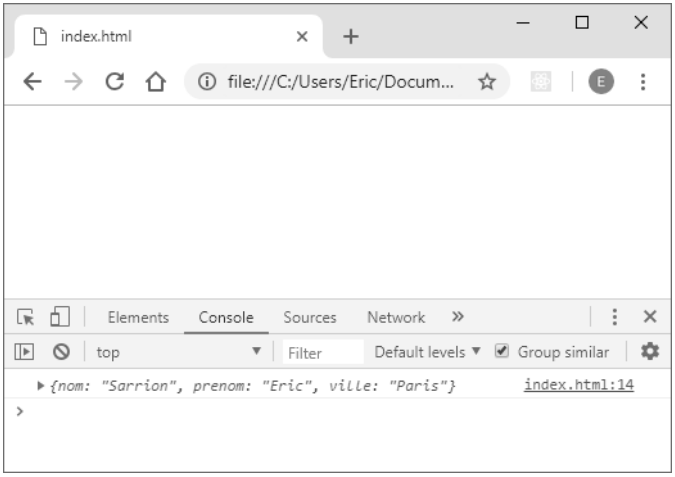
Cela était déjà possible avec les versions antérieures de JavaScript, mais nécessitait une syntaxe plus verbeuse. Pour créer un objet `personne` à partir des variables `nom`, `prenom` et `ville` définies dans le programme, on écrivait par exemple ce qui suit.

Définir un objet personne à partir des variables nom, prenom et ville

```
var nom = "Sarrion";
var prenom = "Eric";
var ville = "Paris";

var personne = { nom : nom, prenom : prenom, ville : ville };
console.log(personne);
```

L'objet `personne` possède les propriétés `nom`, `prenom` et `ville` qui correspondent aux mêmes noms que les variables définies dans le programme, d'où la redondance entre le nom de la propriété (à gauche du caractère `:`) et sa valeur (à droite du caractère `:`).

Figure 1–17



Plutôt que d'écrire l'objet `personne` sous la forme `propriété : valeur`, ES6 permet d'écrire uniquement le nom de la propriété, sans la valeur, car celle-ci correspondra à la valeur de la variable ayant le même nom que la propriété. On écrira donc, en utilisant la structuration des objets ES6, le code suivant.

Définir un objet personne en ES6 à partir des variables nom, prenom et ville

```
var nom = "Sarrion";
var prenom = "Eric";
var ville = "Paris";

var personne = { nom, prenom, ville };   // Structuration de l'objet personne
console.log(personne);
```

Les propriétés `nom`, `prenom` et `ville` de l'objet `personne` doivent correspondre à des variables définies au préalable dans le code JavaScript. Ces variables sont ici définies par le mot-clé `var`, mais elles pourraient aussi être définies par les mots-clés `let` ou `const` vus précédemment.

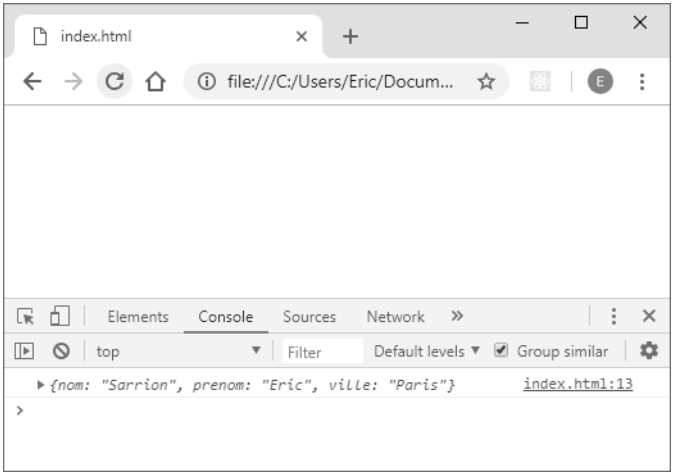
On peut mixer l'ancienne notation (avec le caractère `:`) et la nouvelle. Par exemple, en définissant la propriété `ville` directement dans l'objet `personne` sans passer par une variable `ville`.

Utilisation des deux notations JavaScript

```
var nom = "Sarrion";
var prenom = "Eric";

var personne = { nom, prenom, ville : "Paris" };
console.log(personne);
```

Le `nom` et le `prenom` sont récupérés directement depuis les variables de même nom, tandis que la propriété `ville` est directement affectée dans l'objet `personne`.

Figure 1–18



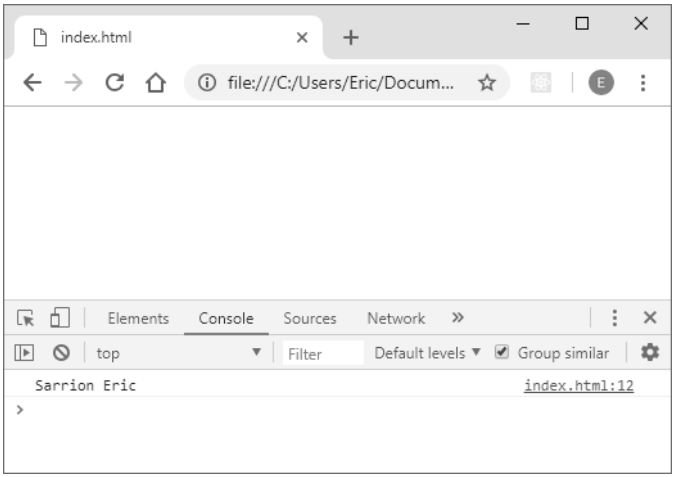
Si une fonction est définie dans les propriétés de l'objet `personne`, on peut l'écrire de la façon suivante.

Définition de la fonction log() dans l'objet personne

```
var nom = "Sarrion";
var prenom = "Eric";
var log = function() {
  console.log(`${this.nom} ${this.prenom}`);
}

var personne = { nom, prenom, log };
personne.log();
```

La fonction `log()` est définie sur les propriétés de l'objet `personne` de la même façon que les autres propriétés de l'objet. Une variable `log` est recherchée et si elle est trouvée (ce qui est le cas ici) sa valeur est attachée à cette propriété de l'objet.

Figure 1–19



La fonction `log()` peut également être définie de la façon suivante (avec la notation =>) :

Fonction log() définie avec =>

```
var nom = "Sarrion";
var prenom = "Eric";
var log = () => {
  console.log(`${this.nom} ${this.prenom}`);
}

var personne = { nom, prenom, log };
personne.log();
```

Une façon plus classique d'écrire l'objet `personne` est la suivante.

Définition de l'objet personne intégrant la fonction log()

```
var nom = "Sarrion";
var prenom = "Eric";

var personne = {
  nom,
  prenom,
  log : function() {
    console.log(`${this.nom} ${this.prenom}`);
  }
};
personne.log();
```

ES6 permet de supprimer le mot-clé `function` lors de la définition de la fonction dans l'objet. On peut alors écrire l'objet de la façon suivante.

Définition de l'objet personne en ES6

```
var nom = "Sarrion";
var prenom = "Eric";

var personne = {
  nom,
  prenom,
  log() {
    console.log(`${this.nom} ${this.prenom}`);
  }
};
personne.log();
```

C'est cette dernière façon de déclarer la fonction dans l'objet qui sera utilisée dans le code JavaScript, ainsi que lors de la définition des classes en React.

Opérateur … sur les objets

L'opérateur … (appelé opérateur *spread*, c'est-à-dire opérateur d'éclatement) permet d'éclater un objet dans ses différentes propriétés. Voyons ce que cela signifie avec un exemple.

On souhaite créer un nouvel objet `personne2` à partir de l'objet `personne`, mais ce nouvel objet `personne2` devra contenir (en plus) la propriété `ville` (qui n'existe pas dans l'objet `personne`).

Créer un nouvel objet à partir d'un objet existant

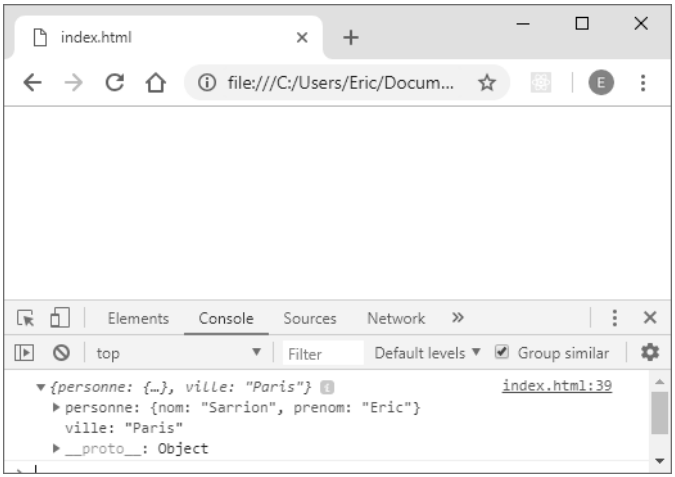
```
var personne = { nom : "Sarrion", prenom : "Eric" };
var ville = "Paris";

var personne2 = {
  personne,
  ville
}

console.log(personne2);
     // { personne : { nom : "Sarrion", prenom : "Eric" }, ville : "Paris" }
```

L'objet `personne` est intégré dans l'objet `personne2`. Mais le résultat n'est pas forcément celui attendu (figure 1-20, page suivante).

L'objet `personne`, intégré dans l'objet `personne2`, est devenu une propriété `personne` de ce nouvel objet. On aurait plutôt voulu que les propriétés de l'objet `personne` s'intègrent directement dans le nouvel objet, de façon à ce que celui-ci ait donc les propriétés `nom`, `prenom` et `ville` (au lieu des propriétés `personne` et `ville` comme c'est le cas actuellement).

L'opérateur … va nous aider à réaliser l'éclatement des propriétés de l'objet `personne` en les intégrant directement dans le nouvel objet. Pour cela, il suffit d'écrire le code qui suit.

Figure 1–20



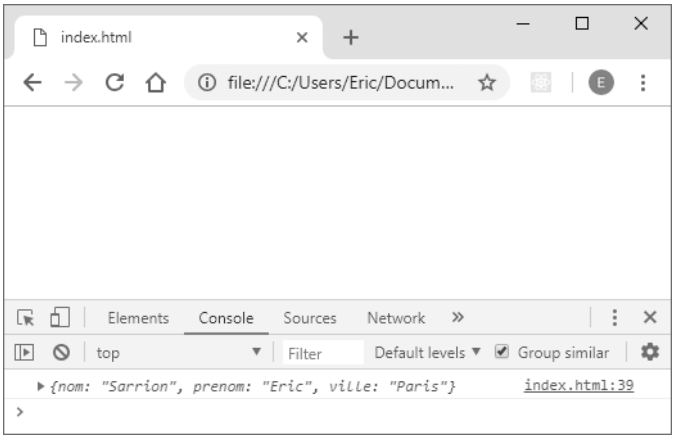
Créer un nouvel objet à partir d'un objet existant en utilisant l'opérateur ...

```
var personne = { nom : "Sarrion", prenom : "Eric" };
var ville = "Paris";

var personne2 = {
  ...personne,   // Objet personne éclaté
  ville
}

console.log(personne2);
```

La seule différence avec le programme précédent est l'opérateur ... utilisé pour éclater les propriétés de l'objet `personne` en les intégrant directement dans le nouvel objet.

L'opérateur ... ne peut s'utiliser que dans un objet JavaScript, comme on peut le voir dans l'exemple précédent.

Figure 1–21



L'objet `personne` a été éclaté et intégré dans le nouvel objet avec ses propriétés directement attachées au nouvel objet.

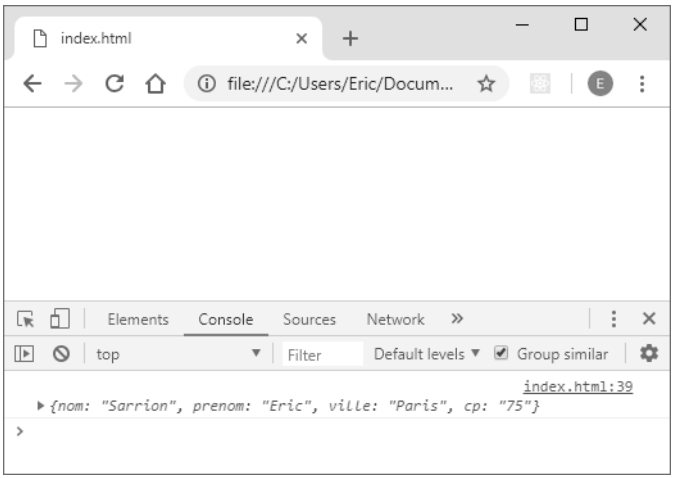
Plusieurs opérateurs `...` peuvent figurer dans la définition d'un objet. Si lors de l'éclatement, des propriétés portent le même nom dans différents objets éclatés, une seule propriété ayant ce nom est insérée dans le nouvel objet (la dernière rencontrée).

Par exemple, on peut créer un nouvel objet à partir des objets `personne` et `place` (l'objet `place` permettant de définir la `ville` et le code postal (propriété `cp`)).

Utilisation de plusieurs opérateurs … pour définir un nouvel objet

```
var personne = { nom : "Sarrion", prenom : "Eric" };
var place = { ville : "Paris", cp : "75" };

var personne2 = {
  ...personne,    // Objet personne éclaté
  ...place        // Objet place éclaté
}

console.log(personne2);
```

Figure 1–22



Les deux objets (`personne` et `place`) ont été éclatés, et leurs propriétés sont attachées à l'objet `personne2` ainsi créé.

Les tableaux

Nous venons de voir qu'il est possible de déstructurer ou structurer un objet. Ce type d'opération peut également être effectué avec les tableaux.