4. Le langage JSX

Les concepteurs de React ont donc cherché un moyen d'alléger l'écriture et leur choix s'est porté sur l'utilisation de JSX (JavaScript eXtension). JSX est une forme d'écriture des éléments React, plus simple à lire et à écrire que les instructions **React.createElement().** Cette syntaxe est donc abondamment utilisée dans les programmes React.

Au fur et à mesure, vous verrez que le JSX est un langage très intuitif à utiliser. Voici les deux propriétés de base pour ce qui est des balises utilisées :

- Toute balise commençant par une minuscule (div, span, label, etc.) est réservé aux éléments HTML. Ces éléments sont déclarés par React DOM, et vous obtiendrez une erreur si vous utilisez un élément inexistant.

- Toute balise commençant par une majuscule (Greetings, App, etc.) doit être déclarée explicitement, ce doit donc être un élément du scope courant : fonction déjà déclarée, composant importé d'une bibliothèque ou d'un autre fichier…

4.1. Hello React avec JSX

JSX est donc une forme nouvelle d'écriture des éléments React. Par exemple, voici un para- graphe contenant Hello React dans son texte.

Un paragraphe contenant Hello React en JSX

```
var p = <p>Hello React</p>; // Code JSX
```

Pour afficher ce paragraphe, on utilise l'outil Babel pour interpréter le code JSX et de le transformer en interne en code JavaScript compréhensible par le navigateur. D'autres outils existent, nous utiliserons celui-ci.

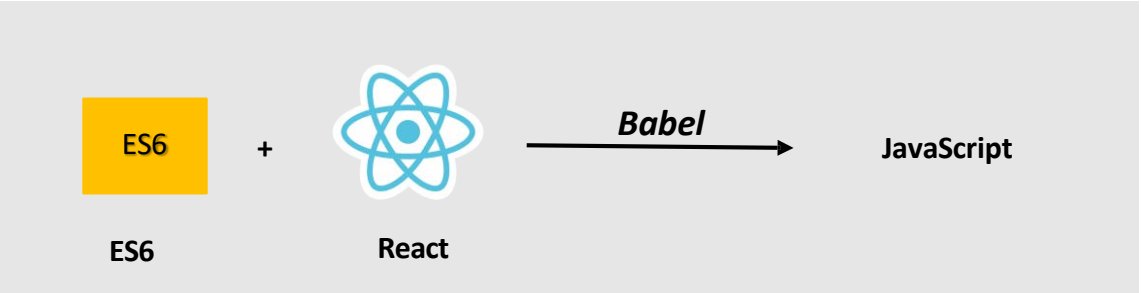
La page HTML utilisant Babel s'écrit de la façon suivante, en intégrant notre code JSX.

Fichier index.html utilisant Babel afin d'interpréter le code JSX.

```
<html>
    <head>
        <script
src="https://unpkg.com/react@16/umd/react.development.js"
crossorigin></script>
        <script src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js" crossorigin></script>
        <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
    </head>
    <body>
        <div id="app"></div>
    </body>
```

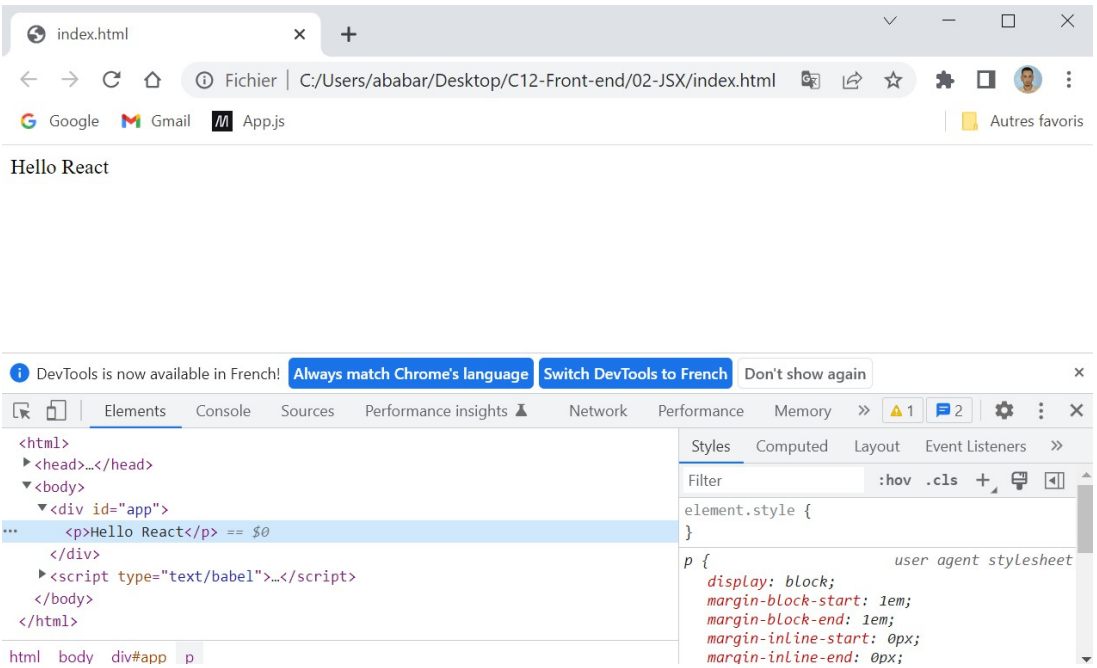
```
    <script type="text/babel">
        var p = <p>Hello React</p>; // Code JSX
        console.log(p); // Affichage dans la console de l'élément
React
        ReactDOM.render(p, document.getElementById("app"));
    </script>
</html>
```

On inclut le fichier JavaScript de **Babel** au moyen de la balise **<script src="...">** (cela cor- respond à l'interpréteur qui traduira le code JSX en code JavaScript), puis on indique quelle partie du code JavaScript est à interpréter par Babel. Pour cela, on inclut l'attribut **type="text/babel"** dans la balise **<script>** contenant notre code JavaScript (et JSX).



Le code JavaScript permettant la création des éléments React est écrit en JSX (et sera traduit en JavaScript pur par Babel), tandis que les éléments React ainsi créés seront insérés dans la page HTML au moyen de l'instruction **ReactDOM.render()**.

L'affichage correspond au paragraphe contenant "Hello React", tandis que l'onglet React de la fenêtre des outils de développement montre l'élément React créé suite à la transformation du code JSX par Babel.

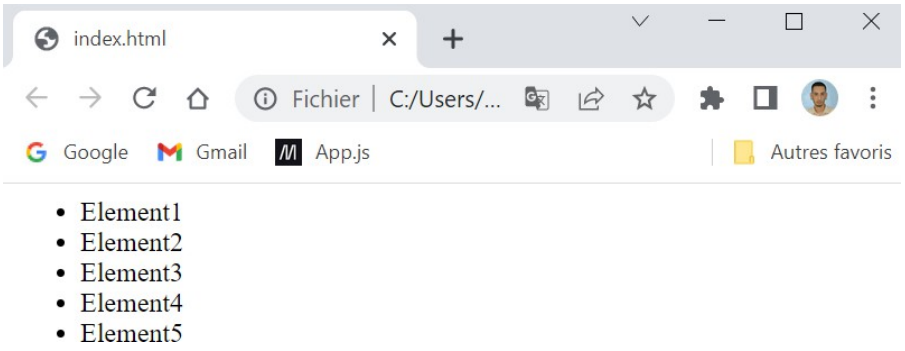


Remarquons que l'inclusion de Babel pour interpréter le code JSX ralentit le programme, vu qu'une étape de traduction est nécessaire avant d'exécuter le code JavaScript. Par conséquent, l'utilisation de Babel ne peut être viable que dans le cadre de l'écriture du programme (en mode développement). Cet outil ne peut pas être utilisé dans le cadre d'un déploiement (mode production). Dans ce dernier cas, on utilisera d'autres outils tels que Webpack pour créer un package plus compact.

Créer une liste de cinq éléments en JSX

```
var liste = <ul>
                <li> Element1 </li>
                <li> Element2 </li>
                <li> Element3 </li>
                <li> Element4 </li>
                <li> Element5 </li>
            </ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

Le code JSX est facile à lire et à écrire. Il s'écrit comme du code HTML, mais il est saisi dans la partie réservée au code JavaScript (dans la balise <script>). Une même instruction peut s'écrire sur plusieurs lignes et doit obligatoirement commencer par une balise ouvrante et se terminer par balise fermante (ici, <ul> et).



Ajouts d'attributs dans le code JSX

Le code JSX comporte les éléments qui seront affichés dans la page HTML. Ces éléments peuvent avoir des attributs tels que id, style ou className (l'attribut class est remplacé par l'attribut className).

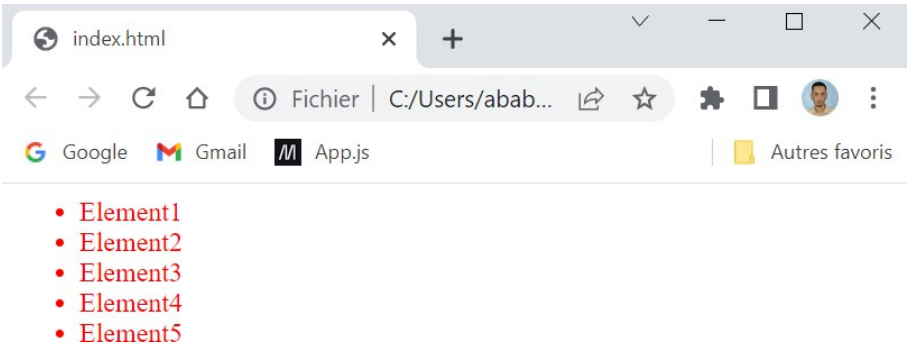
Commençons par ajouter les attributs id et className. Pour cela, on définit la classe CSS red dans la balise <style> de la page.

```
<style type="text/css">
    .red {
    color : red;
    }
</style>
```

Définir les attributs id et className dans le code JSX

```
var liste = <ul id="list1" className="red">
                <li> Element1 </li>
                <li> Element2 </li>
                <li> Element3 </li>
                <li> Element4 </li>
                <li> Element5 </li>
            </ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

La liste définie par <ul> possède bien l'id "list1", tandis que la classe CSS red est bien définie sur la liste (les éléments de liste sont de couleur rouge).



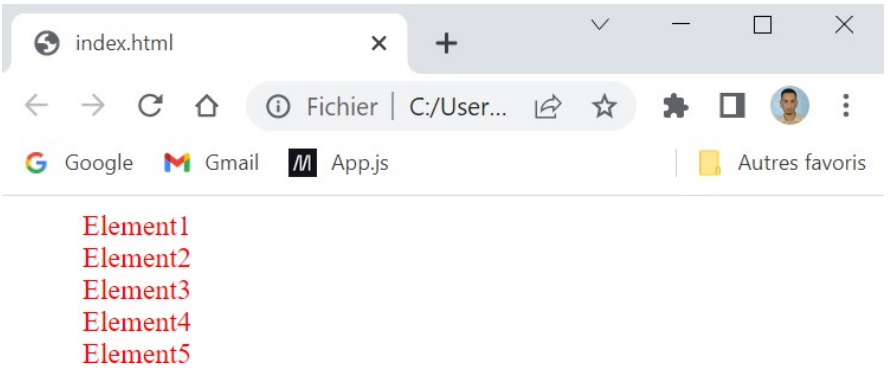
Ajout de l'attribut style en JSX

La syntaxe à utiliser pour insérer l'attribut style est légèrement différente. On souhaite maintenant définir dans le style de la liste (élément <ul>) la propriété CSS list-style-type et lui attribuer la valeur "none", ce qui signifie que les éléments de liste s'affichent sans être précédés par un point. La propriété color sera également définie dans le style à la valeur "red" (on suppose que l'on enlève l'attribut className utilisé précédemment de façon à ce que la couleur des éléments de liste ne soit pas définie à deux endroits).

Définir l'attribut style dans le code JSX

```
var liste = <ul id="list1" style={{listStyleType:"none",
color:"red"}}>
                <li> Element1 </li>
                <li> Element2 </li>
                <li> Element3 </li>
                <li> Element4 </li>
                <li> Element5 </li>
            </ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

La propriété list-style-type s'écrit dans le code JavaScript sous la forme listStyleType, en remplaçant comme d'habitude chaque tiret et la lettre qui le suit par une majuscule. L'attribut style est défini en utilisant les caractères **{{** et **}}**, soit deux accolades ouvrantes puis deux fermantes. Les accolades extérieures indiquent que l'expression à l'intérieur est une expres- sion JavaScript. De plus, le style doit dans ce cas être défini au moyen d'un objet JavaScript.



4.2. Utilisation d'instructions JavaScript dans le code JSX

On peut utiliser des instructions JavaScript dans du code JSX, à condition d'entourer les instructions JavaScript avec des accolades. Chaque instruction entourée d'accolades est évaluée par le navigateur, et son résultat est inséré en lieu et place de l'instruction JavaScript évaluée. Ceci permet de créer du code JSX qui s'adapte aux conditions définies dans le programme.

Définir des instructions JavaScript qui calculent le style de l'élément en JSX

```
var color = "red";
var styleListe = { listStyleType:"none", color:color };
var liste = <ul id="list1" style={styleListe}>
                <li> Element1 </li>
                <li> Element2 </li>
                <li> Element3 </li>
                <li> Element4 </li>
                <li> Element5 </li>
            </ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

L'instruction JavaScript {styleListe} indique de calculer la valeur de l'expression styleListe, puis d'affecter cette valeur au style de l'élément dans le code JSX.

Insérer les éléments de liste définis dans un tableau elems

On peut améliorer notre code en insérant les éléments de liste au moyen d'un bloc de code JavaScript. Les éléments de la liste sont placés dans un tableau elems qui est ensuite parcouru par le code JavaScript et JSX.

```
var color = "red";
var styleListe = { listStyleType:"none", color:color };
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var liste = <ul id="list1" style={styleListe}>
                {
                    elems.map(function(elem, index) {
                        return <li key={index}>{elem}</li>
                    })
                }
            </ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

Les instructions JavaScript dans un bloc de code JSX doivent être entourées par des acco- lades, en particulier l'instruction **elems.map().** De même, dans chaque instruction JSX, toute expression JavaScript doit être encadrée par des accolades, d'où leur présence dans **{styleListe}** et **{index}**.

L'attribut key est similaire à celui utilisé dans le précédent chapitre et permet d'éviter un aver- tissement lors de l'exécution du code (message d'erreur «Each child in an array or iterator should have a unique "key" prop.»).

Utiliser la notation ES6 pour définir la fonction

En utilisant la notation => (disponible dans ES6) pour définir la fonction de callback, on peut écrire plus simplement le code suivant.

```
var color = "red";
var styleListe = { listStyleType:"none", color:color };
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var liste = <ul id="list1" style={styleListe}>
                {
                    elems.map((elem, index) => {
                        return <li key={index}>{elem}</li>
                    })
                }
            </ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

Utiliser la notation ES6 sans accolades ni instruction return dans la fonction de callback

Ce qui peut aussi s'écrire de façon encore plus raccourcie (les accolades et l'instruction return dans la fonction de callback ne sont pas nécessaires si une seule instruction est présente dans les accolades).

```
var color = "red";
var styleListe = { listStyleType:"none", color:color };
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var liste = <ul id="list1" style={styleListe}>
                {
                    elems.map((elem, index) =>
                       <li key={index}>{elem}</li>
                    )
                }
            </ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

4.3. Créer un élément JSX avec une fonction

L'intérêt de JSX est qu'il permet de créer ses propres éléments HTML, qui seront vus comme des éléments React (écrits en JSX). On va donc ici apprendre à créer l'élément **<ListeElements>** qui représentera la liste **<ul>** contenant les éléments **<li>.**

Créer une fonction qui retourne du code JSX

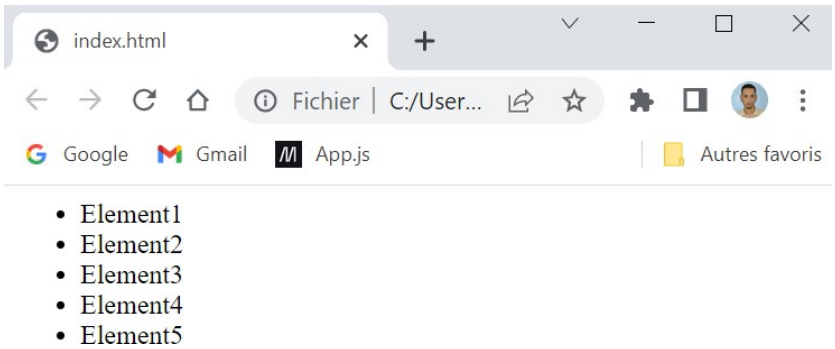
Améliorons le précédent programme pour le transformer en une fonction qui retourne le code JSX nécessaire à la création de la liste. Dans le chapitre précédent, nous avions réalisé une fonction similaire, mais qui retournait la liste au moyen des instructions **React.createElement()**. Ici, nous n'utilisons pas ces instructions mais plutôt le code JSX.

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function() {
    return <ul>
        {
            elems.map(function(elem, index) {
                return <li key={index}>{elem}</li>;
            })
        }
    </ul>
}
ReactDOM.render(<ListeElements/>, document.getElementById("app"));
```

La méthode **ReactDOM.render()** prend ici en premier argument un élément React défini en JSX (<ListeElements/>). Cet élément correspond à une fonction du même nom qui crée et retourne les éléments React définis également en JSX.

Remarquez qu'un élément défini en JSX, tel que **<ListeElements/>**, doit obligatoirement com- mencer par une majuscule, sinon React produit une erreur. La fonction associée correspondante doit donc également commencer par une majuscule. Les seuls éléments JSX pouvant commencer par une minuscule sont ceux correspondants à des balises HTML, telles que <ul>, <li>, etc.

On voit dans l'onglet React qu'un élément React nommé <ListeElements> a été créé par React, et qu'il contient la liste définie par <ul>.



Retourner uniquement les éléments <li> dans la fonction (sans l'élément <ul>)

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function() {
    return elems.map(function(elem, index) {
                return <li key={index}>{elem}</li>;
            })
}
ReactDOM.render(<ul><ListeElements/></ul>,
document.getElementById("app"));
```

La fonction ne retourne plus l'élément <ul>, donc les accolades qui servaient à indiquer le code JavaScript à l'intérieur du code JSX ne sont ici plus nécessaires (et si vous les laissez, elles provoquent une erreur).

En revanche, la méthode **ReactDOM.render()** doit retourner le code JSX complet, incluant l'élément <ul>.

Même si l'affichage de la liste est identique au précédent, on voit ici que les éléments React <ul> et <ListeElements> ont été inversés dans l'arborescence.

Transmettre des attributs dans un élément JSX

On peut transmettre des attributs aux éléments React définis ici en JSX. Par exemple, le tableau **elems** pourrait être transmis dans l'attribut **elems** de l'élément JSX. On peut créer les attributs que l'on souhaite dans un élément JSX, ces attributs seront transmis en paramètres de la fonction de traitement dans l'objet **props**.

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function(props) {
    return <ul>
            {
                props.elems.map(function(elem, index) {
                    return <li key={index}>{elem}</li>;
                })
            }
            </ul>
}
ReactDOM.render(<ListeElements elems={elems}/>,
document.getElementById("app"));
```

L'attribut **elems** est défini lors de l'écriture de l'élément JSX **<ListeElements elems={elems}/>**. Les attributs d'un élément défini par une fonction sont transmis dans l'objet **props** en paramètres de la fonction. Ainsi, pour accéder à l'attribut elems dans la fonction, on utilise props.elems.

Transmission de l'attribut style dans l'élément JSX

Transmettons maintenant l'attribut style dans l'élément JSX. Le style indiqué sera affecté aux éléments <li> de la liste.

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function(props) {
    return <ul>
            {
                props.elems.map(function(elem, index) {
                    return <li key={index}
style={props.style}>{elem}</li>;
                })
            }
            </ul>
}
ReactDOM.render(<ListeElements elems={elems}
style={{color:"red"}}/>, document.getElementById("app"));
```

Le style est indiqué comme d'habitude sous forme d'objet JSON (ici**, { color:"red" }**), et comme c'est une instruction JavaScript, il faut l'entourer des accolades, d'où les doubles acco- lades que l'on peut voir ici dans l'élément JSX.

Ce style est récupéré dans la fonction au moyen du paramètre props, et il est accédé à l'aide de props.style dans l'élément JSX définissant chaque élément <li>.

Écriture du programme en déstructurant l'objet props (en ES6)

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function({elems, color}) {
    return <ul>
            {
                elems.map(function(elem, index) {
                   return <li key={index} style={{color:color}}>
                   {elem}</li>;
                })
            }
            </ul>
}
ReactDOM.render(<ListeElements elems={elems} color="red"/>,
document.getElementById("app"));
```

On accède maintenant directement aux variables **elems** et **color** précédemment définies comme propriétés dans l'objet **props**.

Créer la liste au moyen de composants

Un élément JSX créé par notre programme est également appelé par un composant React. Ici, le composant est <**ListeElements**> qui représente la liste des éléments à afficher sous forme de liste.

Toutefois, React encourage d'aller plus loin, et de créer un maximum de composants dans nos programmes React. En effet, le but est d'écrire des composants indépendants qui pourront être utilisés à divers endroits du programme, voire dans d'autres programmes. Cela permet la modularité et la réutilisation du code grâce aux composants.

Dans notre programme, il n'est pas difficile de trouver un nouveau composant à écrire. Il pourrait s'appeler <**Element**> et correspondrait à un élément de la liste. Cela correspond à la philosophie de React qui consiste à organiser le code en différents composants qui s'utilisent les uns avec les autres. Le composant principal <**ListeElements**> est donc fait de plusieurs composants <**Element**>.

Écrivons le composant <**Element**> utilisé par le composant <**ListeElements**>.

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var Element = function({color, elem}) {
    return <li style={{color:color}}>{elem}</li>;
}
var ListeElements = function({elems, color}) {
    return <ul>
        {
            elems.map(function(elem, index) {
                return <Element key={index} elem={elem}
color={color} />
            })
        }
        </ul>
}
ReactDOM.render(<ListeElements elems={elems} color="red" />,
```

Le composant <Element> est lui aussi créé avec une fonction dans laquelle les attributs index, color et elem sont transmis en paramètres dans l'objet props (ici, utilisé sous forme déstruc- turée). L'attribut key est utilisé pour éviter l'erreur classique de React indiquant que cet attribut est obligatoire. Toutefois, il ne sert qu'à mettre une clé différente sur les éléments issus d'une fonction d'itération, donc il est utilisé dans l'écriture de l'élément <Element> (écrit dans une boucle d'itération), mais pas dans les paramètres de la fonction Element().

4.4. Créer un élément JSX avec une classe

Dans la section précédente, nous avons vu comment créer un élément JSX à partir d'une fonction. Mais on sait que l'on peut également créer des éléments React (et JSX) à partir d'une classe dérivant de la classe **React.Component**.

Créons maintenant deux classes correspondant aux deux composants utilisés précédemment (<**Element**> et <**ListeElements**>). Ces deux classes dérivent de la classe **React.Component**.

Créer les classes associées aux composants <**Element**> et <**ListeElements**>

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
class Element extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return <li
style={{color:this.props.color}}>{this.props.elem}</li>
    }
}
class ListeElements extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return <ul>
                {
                    this.props.elems.map((elem, index) => {
                        return <Element key={index} elem={elem}
color={this.props.color} />
                    })
                }
            </ul>
    }
}
ReactDOM.render(<ListeElements elems={elems} color="red" />,
document.getElementById("app"));
```

L'instruction **ReactDOM.render()** est la même que celle utilisée dans la section précédente. On transmet dans la classe **ListeElements** les attributs elems et color, utilisés dans la classe via l'objet **this.props** qui les contient.

Remarquez que la fonction de callback utilisée dans la méthode **map()** est définie via la notation ES6 (avec **=>** au lieu de **function**), ceci afin de ne pas perdre la valeur de l'objet **this** dans la fonction de callback (this.props peut donc être accessible dans la fonction de call- back afin que sa propriété color soit utilisée).

Dans la classe **Element**, remarquez l'utilisation des doubles accolades pour définir le style: la première paire d'accolades est utilisée pour indiquer une instruction JavaScript, la seconde est utilisée pour écrire l'objet sous forme JSON.

On voit sur l'exemple précédent l'utilité de la notation des fonctions en ES6 (avec les caractères =>) qui évite de perdre la valeur de this dans une fonction de callback. Toutefois, on peut écrire le programme de façon légèrement différente et ne pas perdre la valeur de this tout en utilisant le mot-clé function pour la fonction de callback.

4.5. Utiliser une fonction ou une classe pour créer les composants en JSX ?

Cette question se pose car les deux manières vues précédemment sont similaires et abou- tissent visiblement aux mêmes résultats.

Comme lorsque l'on s'était posé la question au sujet de la création des fonctions ou des classes avec les éléments React (par **React.createElement()** dans le chapitre précédent), la réponse est similaire:

- on utilisera une fonction si l'on n'a pas besoin de créer des propriétés ou des méthodes pour faciliter les traitements;
- on utilisera plutôt une classe si des propriétés ou des méthodes sont nécessaires pour les traitements.

En fait, une propriété très importante d'un composant sera la propriété state, permettant de gérer l'état du composant (ceci est étudié dans le chapitre suivant). La règle observée est que si le composant possède un état, on utilisera une classe pour le définir (c'est même dans ce cas obligatoire), sinon une fonction sera suffisante.

4.6. Règles d'écriture du code JSX

Un seul élément parent peut être retourné

Plusieurs éléments JSX de même niveau ne peuvent pas être retournés simultanément, il est obligatoire qu'ils soient encapsulés dans un élément parent, qui sera celui retourné (pour être unique), les autres éléments étant ses enfants. En général on utilise un élément englobant l'ensemble, mais React propose aussi d'utiliser un composant <React.Fragment> jouant ce rôle.

Remarque : Cette règle est valable également si on utilise la méthode **React.createElement(),** avec laquelle on doit retourner également un seul élément React parent.

Utiliser un fragment avec le composant <React.Fragment>

L'ajout d'un parent, tel qu'un élément ** ,** fonctionne lorsqu'on souhaite encapsuler plusieurs éléments retournés dans un seul. L'inconvénient de cette solution est que cela ajoute un élément ** ** supplémentaire dans le code JSX, sans que cela soit vraiment nécessaire pour l'application React.

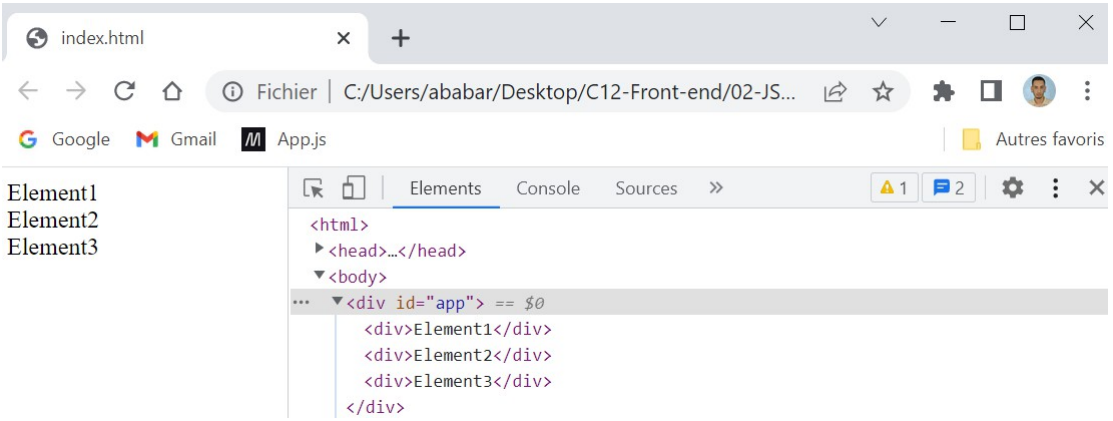
Pour cela, React propose un composant spécifique appelé <**React.Fragment**> que l'on peut utiliser pour ces cas-là.

Utilisons le composant <**React.Fragment**> pour englober un ensemble de trois éléments sans parents. L'élément <**React.Fragment**> va devenir le parent des trois éléments , sans apparaître pour autant dans l'arborescence des éléments React.

```
function ListeElements(props) {
    return <React.Fragment>
        <div>Element1</div>
        <div>Element2</div>
        <div>Element3</div>
    </React.Fragment>
}
ReactDOM.render(<ListeElements />,
document.getElementById("app"));
```

L'élément <**React.Fragment**> permet de retourner un seul parent, en évitant l'ajout d'un nouvel élément parent non nécessaire.

Remarquez que React ne visualise pas l'élément <**React.Fragment**> dans l'arborescence des éléments React.



Utiliser des parenthèses en début et en fin du code JSX

Lorsqu'on retourne un code JSX sur plusieurs lignes (par exemple un élément <ul> suivi de plusieurs éléments <li>), l'instruction return doit comporter à la suite, sur la même ligne, le premier élément JSX retourné, sinon une erreur se produit. Cela oblige à décaler vers la droite le code JSX du premier élément retourné.

Afficher une liste d'éléments sans utiliser des parenthèses

```
function ListeElements(props) {
    return <ul>
            <li>Element1</li>
            <li>Element2</li>
            <li>Element3</li>
            <li>Element4</li>
            <li>Element5</li>
        </ul>
}
ReactDOM.render(<ListeElements />, document.getElementById("app"));
```

Afficher une liste d'éléments en utilisant des parenthèses

```
function ListeElements(props) {
    return (
        <ul>
            <li>Element1</li>
            <li>Element2</li>
            <li>Element3</li>
            <li>Element4</li>
            <li>Element5</li>
        </ul>
    )
}
ReactDOM.render(<ListeElements />, document.getElementById("app"));
```

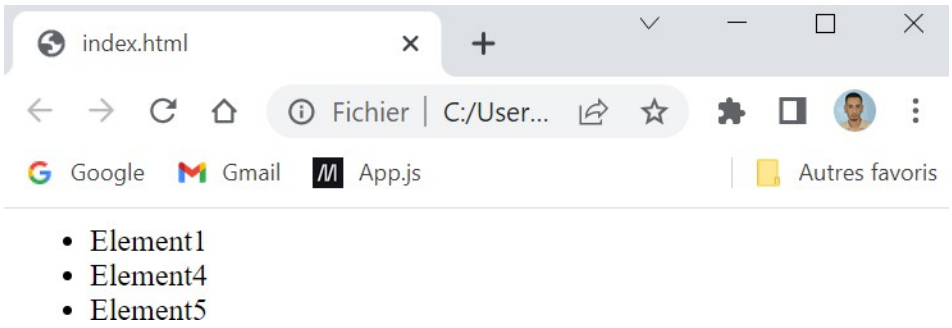
Commentaires dans le code JSX

On utilise les commentaires /* et */ pour indiquer respectivement le début et la fin du code JSX à commenter, à la condition d'entourer l'ensemble avec des accolades { et }.

Les commentaires avec // ne fonctionnent pas avec le code JSX… Par exemple, mettons en commentaires les "Element2" et "Element3" de la liste précédente.

```
function ListeElements(props) {
    return (
        <ul>
            <li>Element1</li>
            {/* <li>Element2</li>
            <li>Element3</li>*/}
            <li>Element4</li>
            <li>Element5</li>
        </ul>
    )
}
ReactDOM.render(<ListeElements />, document.getElementById("app"));
```

Dans les deux exemples de programmes, les éléments mis en commentaires n'apparaissent pas à l'affichage.



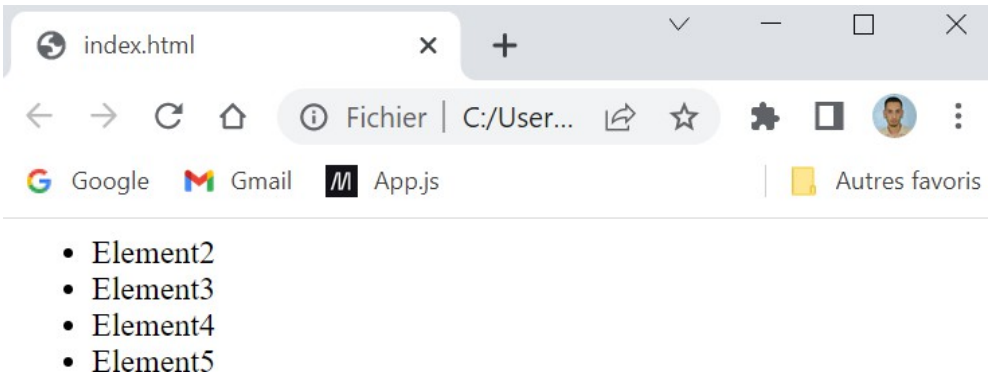
Les éléments "Element2" et "Element3" ne sont pas affichés.

Utiliser des expressions conditionnelles dans le code JSX retourné

Il est possible d'utiliser des expressions conditionnelles avec ? et : dans le code JSX, à condition d'entourer l'ensemble avec des accolades { et } (car cela correspond à une expression JavaScript qui est évaluée).

Supposons que l'on ait un attribut dans le composant <ListeElements> permettant d'indiquer si l'on doit cacher ou pas le premier élément de la liste. L'attribut se nommera hideFirstItem et vaut true si l'on doit cacher cet élément, false sinon.

```
function ListeElements(props) {
    return (
        <ul>
            { props.hideFirstItem ? null : <li>Element1</li> }
            <li>Element2</li>
            <li>Element3</li>
            <li>Element4</li>
            <li>Element5</li>
        </ul>
    )
}
ReactDOM.render(<ListeElements hideFirstItem={true} />,
document.getElementById("app"));
```



Le premier élément de la liste n'apparaît pas.