

React

Eh msieur là haut, vous auriez pas des Props svp?

Dans la quête précédente, nous avons vu que nous pouvions construire une page grâce à React et au JSX en utilisant plusieurs composants qui créent des éléments directement ou appellent d'autres composants.

Les composants qui appellent d'autres composants sont des **composants parents**, et les composants appelés sont les **composants enfants**.


Le fichier index.js n'appelle lui généralement qu'un composant : le composant App.js, qui est le premier composant parent.

Les composants utilisent le JSX donc pour créer des éléments HTML et rendre d'autres composants. Mais le JSX peut aussi utiliser des expressions javascript pour afficher le contenu de **variables** dans les éléments HTML à créer, appeler des **fonctions javascript**, faire des **boucles** ou encore utiliser des **conditions** (avec l'opérateur ternaire). Cela nous permet d'afficher des données dynamiques dans nos composants (titre et contenu d'un article issus d'une base de données) ou encore de faire un rendu conditionnel (créer un élément ou pas en fonction d'une valeur de variable).

Pour pouvoir utiliser des expressions javascript en JSX, on utilise l'**interpolation** grâce aux accolades : « **{** » pour commencer à utiliser du js dans le JSX et « **}** » pour terminer.

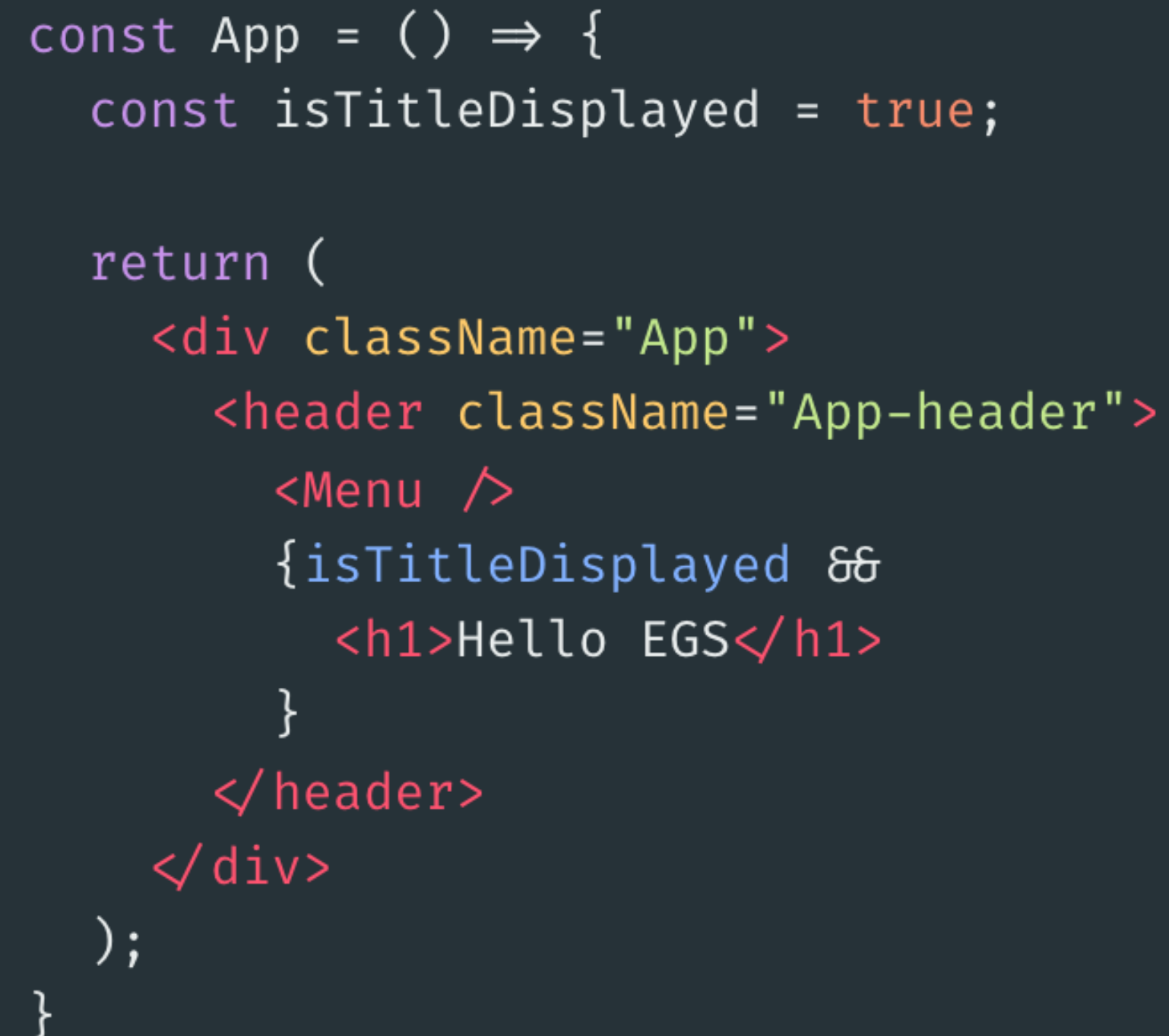
Pour afficher le contenu d'une variable (contenant une chaîne de caractères par exemple), il faudra donc utiliser les accolades dans l'élément JSX, avec le nom de la variable. Attention, les variables sont à créer bien sûr avant le return de la fonction.

Création et affichage d'un titre :



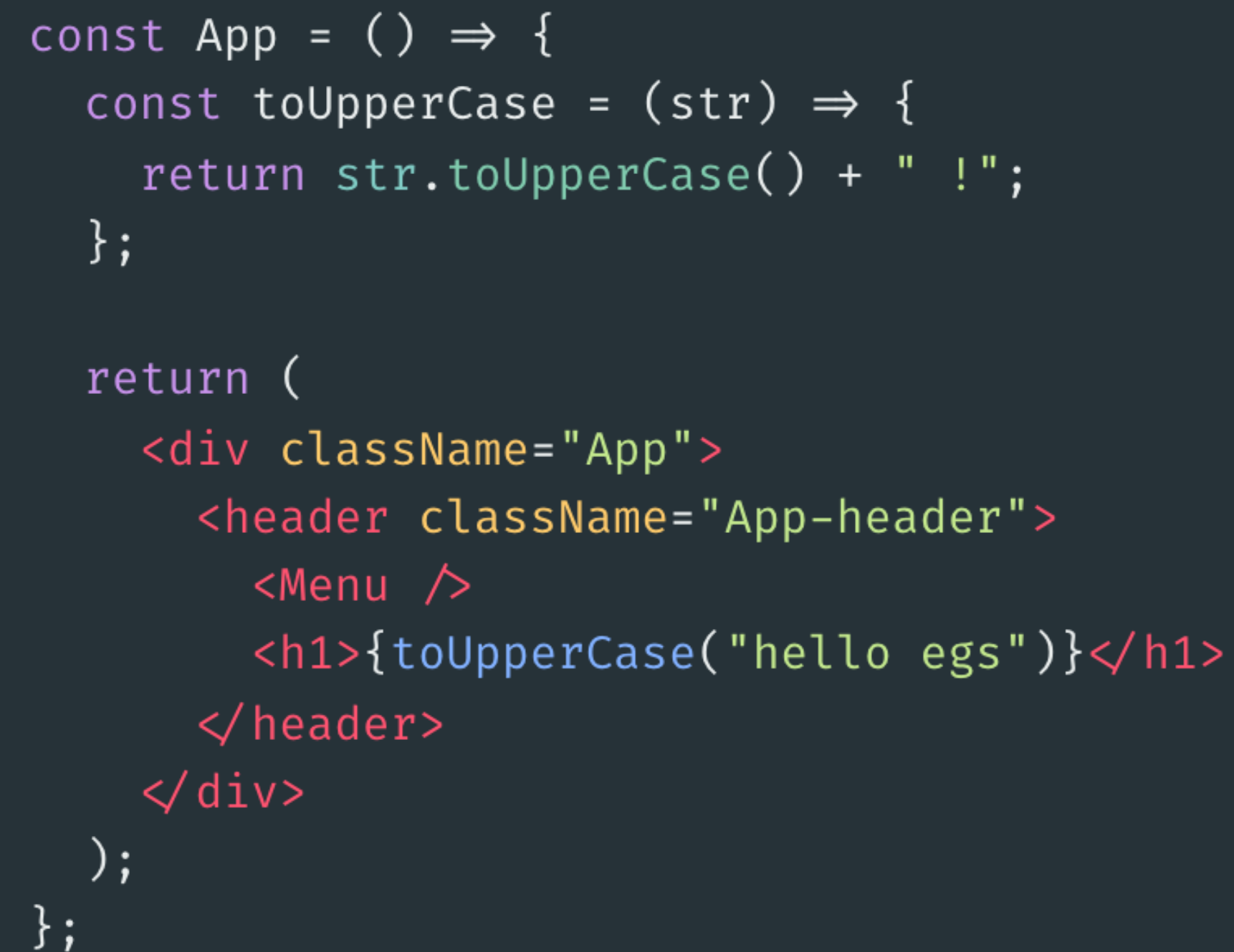
```
function App() {  
  const title = "Bonjour EGS";  
  
  return (  
    <div className="App">  
      <header className="App-header">  
        <Menu />  
        <p>{title}</p>  
      </header>  
    </div>  
  );  
}
```

Pour afficher un élément en fonction d'une condition, il faut utiliser l'opérateur ternaire avec des accolades dans le JSX :



```
const App = () => {  
  const isTitleDisplayed = true;  
  
  return (  
    <div className="App">  
      <header className="App-header">  
        <Menu />  
        {isTitleDisplayed &&  
          <h1>Hello EGS</h1>  
        }  
      </header>  
    </div>  
  );  
}
```

Si on veut appeler une fonction qui, par exemple, prend une chaîne de caractères, la passe en majuscule et lui ajoute un « ! » à la fin :



```
const App = () => {  
  const toUpperCase = (str) => {  
    return str.toUpperCase() + " !";  
  };  
  
  return (  
    <div className="App">  
      <header className="App-header">  
        <Menu />  
        <h1>{toUpperCase("hello eggs")}</h1>  
      </header>  
    </div>  
  );  
};
```

Et enfin, si on veut afficher les éléments d'une liste, on utilisera généralement la méthode **map()** plutôt que la boucle `forEach()`. Les deux permettent de faire une boucle sur un tableau (ou un itérable) à la différence que `map()` crée un nouveau tableau, ce qui permet de ne pas modifier le tableau original (principe important en programmation fonctionnelle) :

```
const App = () => {
  const menuItems = ["Accueil", "Recettes", "Categories", "Contact"];

  return (
    <div className="App">
      <header className="App-header">
        {menuItems.map((item) => {
          return <a>{item}</a>;
        })}
      </header>
    </div>
  );
};
```

A noter que pour que React puisse correctement la modification des listes (suppression d'un élément par exemple, il faut ajouter l'attribut `key` avec en valeur un identifiant unique pour chaque élément dans la boucle

A noter que lors de l'utilisation d'une boucle `map()`, il faut ajouter **l'attribut `key`**, contenant en valeur un identifiant unique, sur chaque élément créé dans la boucle. En effet quand React, pour afficher le contenu créé en JSX, React va générer le **Virtual DOM** contenant votre liste et il va le comparer au DOM réel pour savoir quelles différences existent (il le compare à une copie du DOM réel pour être plus précis). Cela lui permettra de n'insérer dans le DOM réel que les éléments qui ont changé et ainsi gagner en performance.

Cette étape, qui entre dans le processus de **Réconciliation** de React, utilise donc **un algorithme de comparaison** entre les deux arbres du DOM. Pour faire cette comparaison, il faut qu'il puisse identifier de manière unique les éléments créés avec une boucle. D'où l'attribut `key`. Par exemple :

```
const Menu = ({ menuItems }) => {  
  return (  
    <ul>  
      {menuItems.map((item) => {  
        return <li key={item.id}>{item.title}</li>;  
      })}  
    </ul>  
  );  
};  
  
export default Menu;
```


Les variables définies dans un composant ne peuvent pas être utilisées en dehors de ce composant. Il ne s'agit pas là d'un fonctionnement spécifique à React : c'est la notion de **portée** (de scope) en javascript. En effet une variable n'est disponible que dans le **scope** dans laquelle elle est créée. C'est à dire dans la fonction dans laquelle elle est créée (ou dans le « block » pour les variables créées avec le mot clé « **const** »). Plus d'infos : <https://cdiese.fr/le-scope-des-variables-en-javascript>).

Les composants étant des fonctions, les variables définies dans un composant sont internes à ce composant. Comment faire donc pour transmettre des variables entre composant ?

C'est là où la notion de **Props** intervient. Quand un composant est appelé dans un autre composant, le composant parent qui appelle le composant enfant peut lui transmettre des paramètres. Ce qui semble logique, étant donné que les composants sont des fonctions. Les paramètres transmis à un composant enfant lors de son appel sont nommées des « **props** ». Un composant enfant, grâce aux props passés, pourra donc utiliser des variables définies dans le composant parent.

Imaginons que l'on souhaite créer un composant **Menu** qui affiche un `` avec autant de `` qu'il y a d'éléments de menu. Les éléments du menu à afficher sont eux définis dans une variable créée dans le composant App. Pour afficher le menu, le composant App doit donc appeler le composant Menu et lui transmettre la variable contenant les éléments du menu :

App.js

```
import Menu from "../Menu";

const App = () => {
  const menuItems = ["Accueil", "Recettes", "Categories", "Contact"];

  return (
    <div className="App">
      <header className="App-header">
        <Menu menuItems={menuItems} />
      </header>
    </div>
  );
};

export default App;
```

Menu.js

```
const Menu = (props) => {
  return (
    <ul>
      {props.menuItems.map((item) => {
        return <li>{item}</li>;
      })}
    </ul>
  );
};

export default Menu;
```

Pour passer une Props à un composant enfant, il faut donc lors de l'appel du composant enfant lui passer un **attribut JSX** contenant **une clé et une valeur**. La valeur peut être la valeur issue d'une variable, comme dans l'exemple précédent : la valeur de *menuItems* est le tableau défini avec la variable javascript. On peut passer autant d'attributs que l'on souhaite à un composant enfant et donc autant de props que l'on souhaite.

Ces props sont regroupés ensuite par React dans un **objet unique** que l'on nomme généralement « props », utilisable en paramètre du composant enfant. Cet objet contient toutes les Props passées depuis le composant parent.

Dans l'exemple précédent, on peut donc accéder à la props *menuItems* via : *props.menuItems*.

Généralement, pour faciliter l'utilisation des props dans le composant enfant, on utilise la syntaxe de de-structuration. Cela permet d'accéder directement au clé d'un objet sans avoir à préciser le nom de l'objet.

Dans les paramètres du composant enfant, en passant les accolades suivies des clés des props passées dans le parent, on peut ainsi récupérer la valeur de chaque clé de l'objet props avec : *const Menu = ({menuItems}) => {...}*.

La variable menuItems sera donc directement accessible sans avoir à utiliser l'objet props.

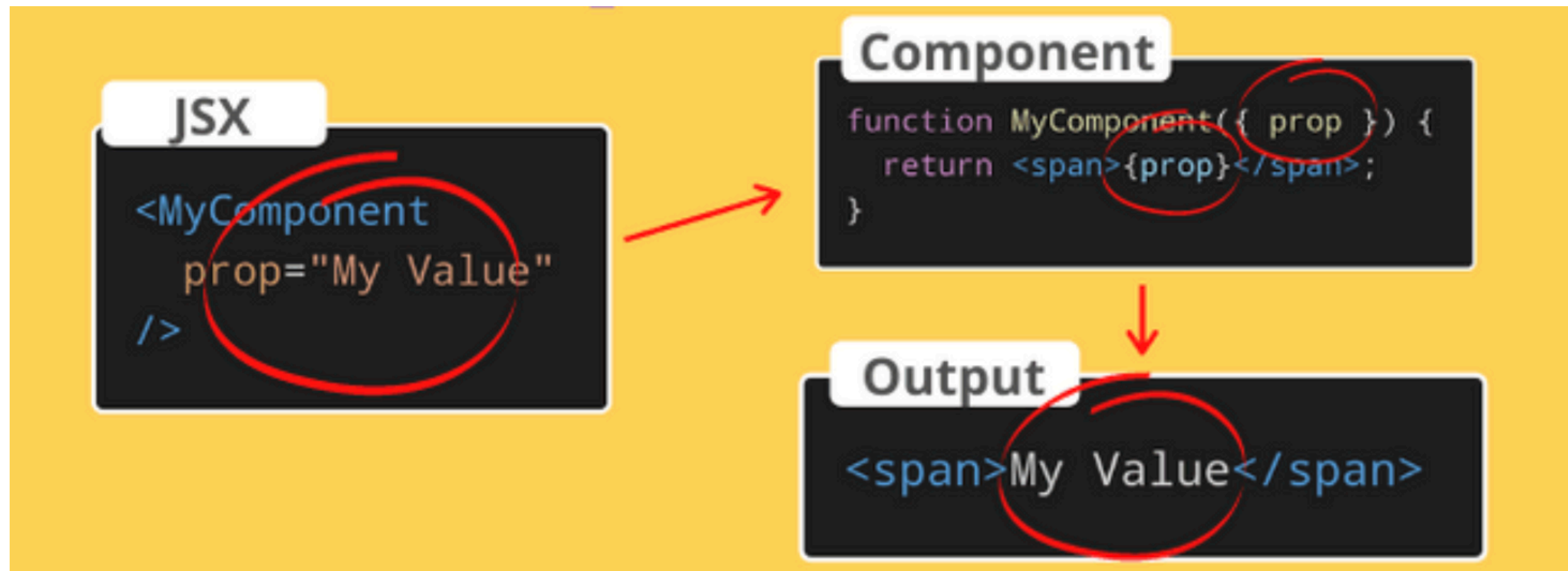
Sans de-structuration :

```
const Menu = (props) => {  
  return (  
    <ul>  
      {props.menuItems.map((item) => {  
        return <li>{item}</li>;  
      })}  
    </ul>  
  );  
};  
  
export default Menu;
```

Avec de-structuration :

```
const Menu = ({menuItems}) => {  
  return (  
    <ul>  
      {menuItems.map((item) => {  
        return <li>{item}</li>;  
      })}  
    </ul>  
  );  
};  
  
export default Menu;
```

Passages des props d'un composant parent à enfant :



Grâce aux props, on peut donc envoyer des données depuis un composant parent vers des composants enfants, qui peuvent les utiliser dans les éléments du DOM créés, ou alors à leur tour transmettre ces données à d'autres composants.

A noter que quand un composant est appelé, c'est à dire « **rendu** » pour la première fois dans le DOM, les composants enfants de ce composant sont eux aussi rendus. Ce qui semble logique : si un composant Header utilise un autre composant Menu pour construire le Header, il faut que les éléments du Menu soient rendus pour que le Header entier puisse être rendu.

Le rendu d'un composant est étroitement lié à la notion de **cycle de vie**. En effet, si un composant n'est rendu qu'une seule et unique fois, l'application n'est pas dynamique. Pour que l'application soit dynamique et que l'affichage puisse être modifié en fonction des actions utilisateurs (click, soumission d'un formulaire etc) il faut que les composants puissent être **re-rendus** quand les données à afficher sont modifiées.

Si les Props envoyées à un composant enfant sont modifiées, le composant enfant utilisant ces props sera ainsi re-exécuté (donc re-rendu) pour tenir compte des changements.