

# REACT JS

React est une bibliothèque, c'est-à-dire une librairie Javascript développée par Facebook en 2013. Elle est open-source, maintenue et enrichie par une très grande communauté.

A quoi sert React Js, et bien c'est utilisé pour la création d'interfaces web mono pages, on peut également créer des applications mobiles multiplateforme grâce à React Native.(cross platform)

Depuis 2019 React a volé des parts aux autres frameworks et ne cesse de monter en flèche depuis. De grande entreprise utilisent React, facebook, instagram, whatsapp, netflix, uber etc...

Nous allons désormais nous lancer et apprendre comme toujours par la pratique, le programme va être découpé en 2 parties :

1. React sans les Hooks c'est-à-dire avant la version 16.8(vous permettra à la fin du programme de pouvoir travailler sur d'ancienne version et la nouvelle) ce qui vous permettra de pouvoir intervenir sur tout React ancien projet et plus récent.
2. React avec les Hooks, ils sont donc apparus dans la version 16.8, une alternative aux classes, tout en comparant avec l'ancienne version. On abordera les useState, (composant de type fonction), les useEffect qui va nous permettre de faire ce que l'on fait avec les classes via les méthodes de cycle de vie d'un composant. Et bien plus encore.

Nous essaierons de voir également l'authentification, les bases de données, et si possible Redux et le déploiement, dans tous les cas se sera vu en React Native. Quelques rappels sur les règles simples de demander de l'aide, on évite les phrases du types :

- « Mon appli ne marche pas »
- J'ai fait tout comme toi, mais j'ai aucun résultat

Mais prenez l'habitude de vérifier dans la console s'il y a un code erreur et si oui entreprenez de faire des recherches sur internet pour essayer de résoudre le souci. Si aucun résultat évidemment je suis là. Ici exceptionnellement on peut consulter une doc bien plus complète que la doc officiel d'origine, vous la trouverez sur [react.dev](https://react.dev) celle-ci inclut des schéma, photos, etc...

## Abordons-les packages NPM

Les packages NPM sont des modules ou des bibliothèques JavaScript qui peuvent être utilisés pour étendre les fonctionnalités de React.js. Ces packages peuvent être installés localement dans un projet React via le gestionnaire de paquets NPM (Node Package Manager).

Il existe de nombreux packages NPM disponibles pour React qui couvrent une variété de fonctionnalités telles que les formulaires, la navigation, les requêtes réseau, la gestion de l'état, etc. On verra quelques-uns de ces package tout au long de cette formation : Par exemple, le package "react-router" qui est très populaire pour la gestion de la navigation dans les applications React ou "Redux" qui, lui, est utilisé pour gérer l'état global de l'application. Les packages NPM peuvent également être utilisés pour ajouter des styles à une application

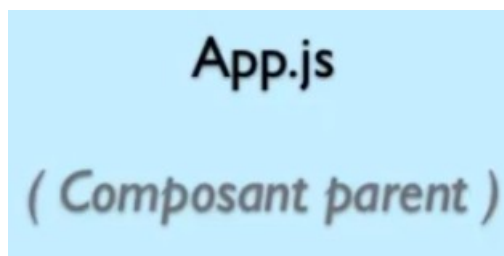
React. Par exemple, le package "styled-components" permet aux développeurs de créer facilement des composants CSS personnalisés pour leur application, etc.

En utilisant des packages NPM, vous pouvez économiser du temps et de l'effort en réutilisant du code existant plutôt que de tout coder vous-mêmes. **Cependant, il est important de s'assurer que les packages choisis sont fiables, bien maintenus et compatibles avec les versions de React utilisées dans le projet.**

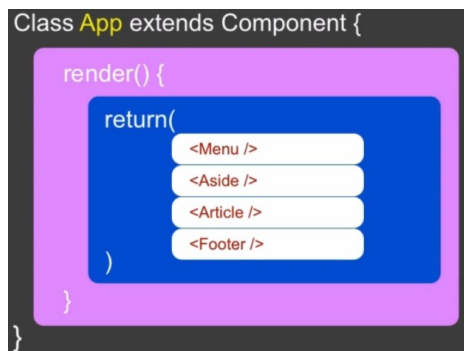
Dans React on va travailler sur le « src », nos packages seront dans le fichier package.json, et nous avons tout face à nous ainsi que les versions, dans la partie **dependencies**. Je vous donnerai les versions à installer pas de panique.

Nous allons donc aborder les composants React(doc reactjs.org) elle est en français, (décomposez la page en composant, header etc.)

On code sur le composant parent de l'application celui qui sera affiché en premier c'est le fichier app.js.



Et l'on va décomposer notre appli de cette manière.



C'est là que l'on va retourner les différents composants que l'on va afficher dans notre application. (ici 4 composants). Ils seront écrits en js et pourront avoir des nested components par exemple header pourrai avoir un composant logo et un composant nav, qui seront importé dans le composant menu.

Dans React nous avons deux types de composants les composants de type fonction et de type classe, le type class est préféré dans le cas où nous souhaitons emmagasiner des données localement dans ce composant afin de les exploiter ensuite. Mais nous verrons cela étape par étape.

Allez, on va coder un peu en React.

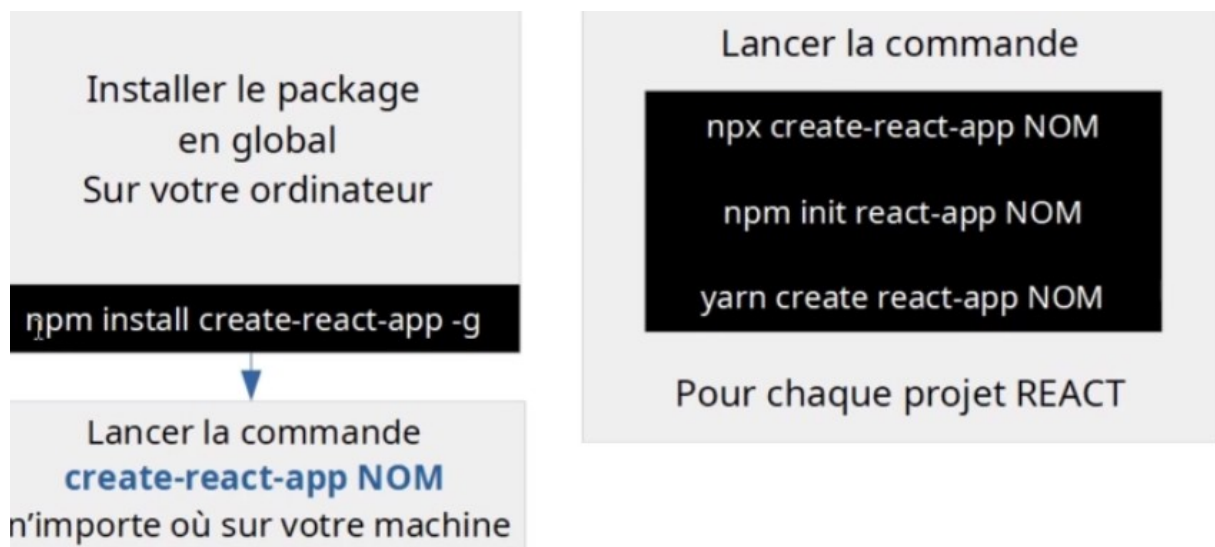
On commence par créer un dossier sur le bureau, appelez le **app voiture** par exemple, et glissez le dans votre ide, allez sur la doc (reactjs.org) et cliquez sur docs puis getting started

sur la droite puis Create a New React App c'est comme cela que nous allons créer nos autres appli par la suite. Pour ce premier exemple vous pouvez utiliser les cdn link si vous le désirez mais il faudra faire un page html pour lire les liens.

Choisissons donc Create a New React App, regardons la doc sur le paragraphe Create React App, et regardons les lignes à taper dans le cli(terminal).

Pour installer en global sur votre ordi, c'est-à-dire si vous comptez créer d'autre app par la suite je vous conseil en global, sinon install à chaque fois.

**NPX CREATE-REACT-APP** étant désormais déprécié, il faut désormais utilisé npx create-next-app@latest



Vous avez la méthode en global et en local désormais. Une fois installé en global je vous conseille de vérifier si tout est correctement installé en faisant ces étapes simples :

**Vérifications :**

```
create-react-app --version
```

```
node --version (node -v)  (≥ 8.10) → nodejs.org/fr
npm --version (npm -v)    (≥ 5.2)
```

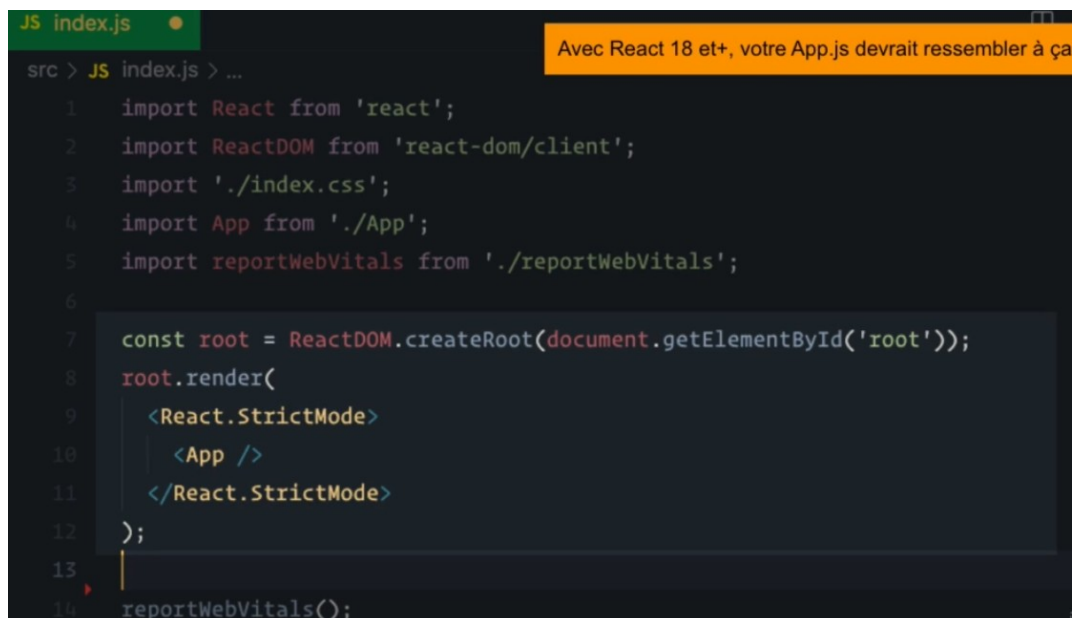
Evidemment pour que l'on puisse créer et travailler nos app React, on va devoir également installer **node** sur notre machine, vous allez à l'adresse mentionné ci-dessus et vous installez cela puis on test avec la ligne version node. (Version 16 minimum)

A vous de jouer on initialise le projet dans l'ide !!!

Vous allez voir que ça va nous installer les dépendances nécessaires, regardez un peu les lignes qu'il vous install, soyez patient.

Alors une fois terminé l'initialisation du projet on peut constater tout ce cela à installer :

1. Node\_modules c'est ici que vont s'installer nos dépendances mais on ne touche surtout pas à ce fichier
2. Public qui contient favicon, index.html et manifest.json. Regardons le index.html on va le nettoyer un peu en virant les commentaires, ensuite regardons la div avec l'id Root, c'est elle ci en fait notre div(on va dire parent de app.js) dans laquelle nous allons afficher les différents composant de notre application
3. Src, qui contient un App.css donc le css de notre appli et app.js et bien c'est le premier composant parent, pour le moment cela peut paraître compliqué mais on va mieux comprendre en codant. Retenez juste que App.js c'est un composant qui contient une fonction qui va nous retourner du Jsx(rien d'extraordinaire c'est comme du html)
4. AppTest.js c'est ici que l'on effectue nos tests on y reviendra plus tard.
5. Index.css bon ben la aussi c'est du css, celui du générique. Ce fichier est utilisé au niveau du fichier suivant index.js, nous avons l'import ici.
6. Index.js



```
JS index.js
src > JS index.js > ...
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3  import './index.css';
4  import App from './App';
5  import reportWebVitals from './reportWebVitals';
6
7  const root = ReactDOM.createRoot(document.getElementById('root'));
8  root.render(
9    <React.StrictMode>
10     <App />
11   </React.StrictMode>
12 );
13
14 reportWebVitals();
```

Le **StrictMode** est un outil que React utilise pour détecter les problèmes potentiels dans votre application. StrictMode n'affiche rien dans votre application, il active simplement des vérifications et avertissements supplémentaires pour ses descendants. Ces vérifications du mode strict sont effectuées uniquement durant le développement. Elles n'impactent pas la version utilisée en production.



```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
reportWebVitals();
```

Petit rappel, en fait dans le fichier index.html nous avons la div avec id Root, l'appli que nous avons dans le App.js nous l'avons exporter avec le 'export default App;', on la récupérer dans index.js, on a bien importer App from ./App dans les imports, une fois importé on utilise le ReactDOM avec la méthode render, et c'est là qu'on spécifie que c'est le composant React

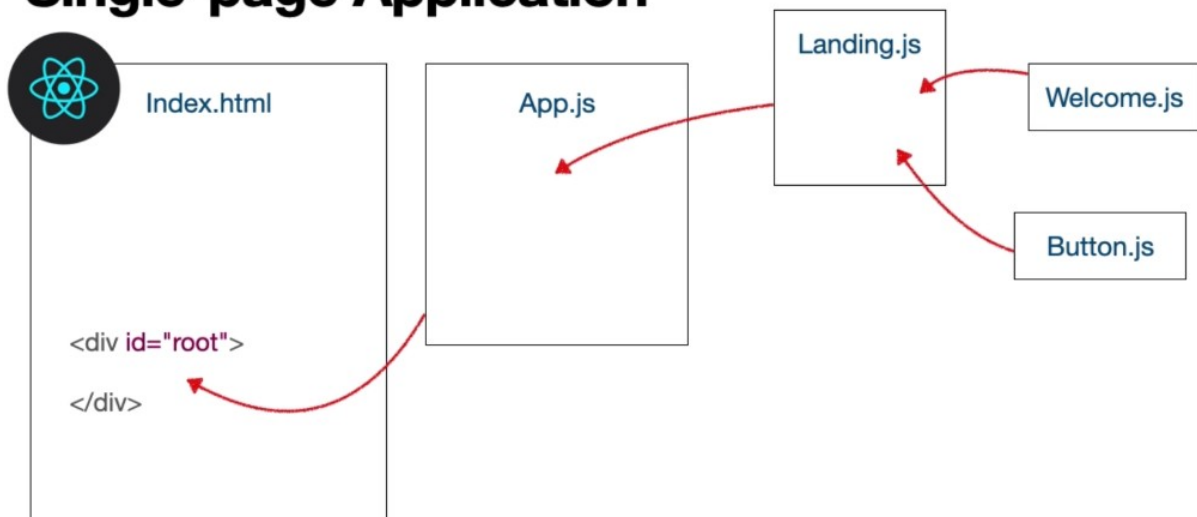
que l'on va afficher dans le document de get element by id dans le Root donc au niveau de index.html voilà. Nous allons coder pas à pas donc pas de panique on reviendra la dessus.

- 7 Package.json c'est là que nous aurons nos fameuses dépendances, a l'initialisation nous avons installé des dépendances on peut les voir, ainsi que les scripts. Parlons du App.js rapidement ainsi que de App.css histoire de mieux comprendre comment ça se passe. Nous avons le fameux « className= App » c'est comme cela que vous écrivez l'équivalent des classe Css en Html, la classe App se trouve bien dans le fichier App.css, nous avons importez notre fichier css avec l'**import**. Pour le logo vous pouvez noter la différence mais on y reviendra plus tard car c'est un composant image. Voilà les différents fichiers de notre appli.

Rendons-nous sur notre terminal et vérifions que l'on est bien sur le bon dossier et on lance le fameux NPM Start, il va compiler et déployer notre application en local sur le port 3000 normalement. Tout va se lancer tout seul vous avez juste à patienter un peu et hop votre première application tourne avec le contenu livré par l'installation, nous allons désormais pouvoir coder tranquillement. Tout ce que vous voyez à l'écran c'est ce qui se trouve dans le App.js le logo, le texte, etc..

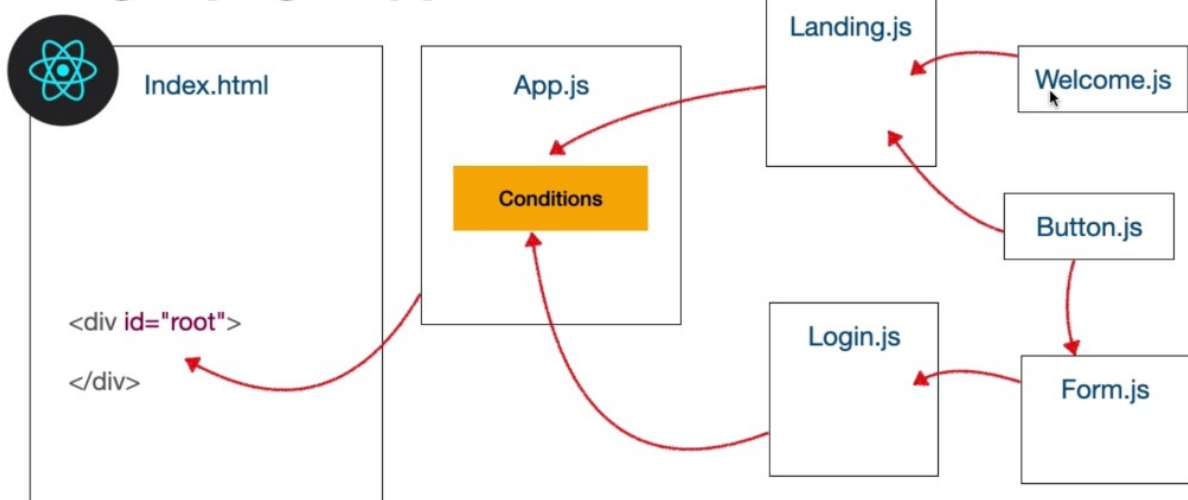
Revenons sur la notion de single page application, je vous invite à vous rendre sur la doc (reactjs.org) on va sur Docs, sur le cote droit vous cliquez sur Référence de l'api, puis glossaire. Petit debrief nous aurons en fait un seul fichier Html, qui contient une div avec id=root et c'est dans cette div la que l'on aura les différents rendus de notre application, comme une porte d'entrée à tout ce que l'on va créer.

## Single-page Application



Voilà comment cela fonctionne, on crée des composants que l'on a externaliser afin de pouvoir les rappeler la ou nous en avons besoin. Si button.js était dans app.js et bien il ne pourrait être utiliser seulement ici, là nous pourrions le rappeler n'importe où.

# Single-page Application



La condition fait que l'on décide ce que l'on veut afficher. C'est ça la logique derrière React si vous avez compris cela et bien vous avez compris la logique de React.

**INFOS** nous avons la méthode `creatRoot()` et l'invocation de la méthode `render()` pour retourner notre app, sachez qu'il y a une alternative pour exactement le même rendu avec un peu moins de code, on peut importer la méthode `creatRoot()` et l'enchaîner avec le `render()`

```
JS index.js
import React from 'react';
import { createRoot } from 'react-dom/client';
import App from './App';
import reportWebVitals from './reportWebVitals';

createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

C'est une méthode que l'on utilise surtout avec les projets en React v18 et plus. Si vous êtes en v17 et que vous souhaitez migrer sur la v18 il suffit de taper dans le terminal :

```
npm install react@18 react-dom@18
```

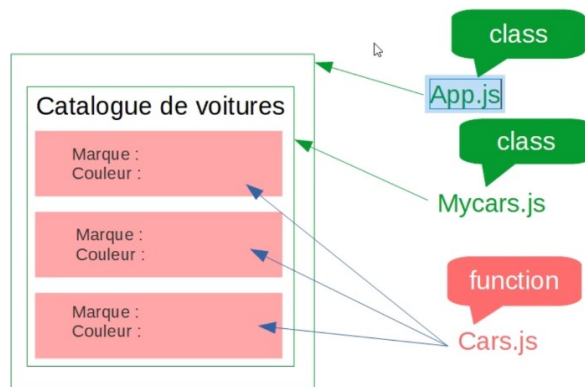
**Allez c'est parti!!!!**

## Les concepts de React

Props et state

Reprenons notre projet `appVoiture`, dans lequel nous avons notre app `Cars`. Dans React nous avons deux types de composants, un composant state (une manière d'enregistrer des données) sous format de classe mais comme je vous ai dit ils veulent les enlever et les fonctions (on fera venir les données d'autres composants grâce aux props). On va commencer par les classes et donc par

logique on aura besoin d'un state qui va nous stocker des data Donc notre objectif est d'arriver à cela.



Nous avons notre composant parent dans lequel nous allons afficher tous les autres composant enfant. Dons dans app.js on va avoir myCars.js qui lui aussi va avoir 3 autres composant cars.js qui eux auront une marque et une couleur.

Prenons l'ide, et enlevons le logo.svg qui ne sert à rien, puis de même sur app.js, pour le moment c'est une fonction que nous allons donc transformer en classe, afin de pouvoir le donner un state. Faisons-le ensemble on remplace donc par une classe et on doit lui donner une methode render() elle a besoin de renvoyer qlq chose nous allons donc lui mettre le return mais dans ce return il y a des chose dont nous n'avons pas besoin.

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <p>
            Edit <code>src/App.js</code> and save to reload.
          </p>
          <a
            className="App-link"
            href="https://reactjs.org"
            target="_blank"
            rel="noopener noreferrer"
          />
        </header>
      </div>
    );
  }
}
```

Nous avons donc désormais une classe, on peut donc regarder le rendu de notre app, on se rend dans le terminal et on lance le script de start(sans le faire vous pouvez me dire où il se trouve et lequel est-ce ?)

Npm start et oui, si erreur à vous de lire l'erreur et de la résoudre.

Vous avez désormais votre appli qui fonctionne, nettoyons tout ce qui ne nous intéresse pas et créons notre app. On a dit on doit créer un composant MyCars, allons dans src, et créons un dossier composants dans lequel on crée un fichier myCars.js , les fichiers ici sont toujours en Majuscule et les dossier en minuscule je vous expliquerai pourquoi plus tard. Donc nous allons créer notre classe on va donc commencer par les imports(extension ES7 REACT/REDUX pour ceux qui ne l'ont pas).



Vous devriez avoir ce résultat,

```
import React, { Component } from 'react';

class Mycars extends Component {
  render() {
    return (
      <div>
        <h1>Hello React</h1>
        <p>ceci est un test</p>
      </div>
    )
  }
}
```

Il faut désormais ajouter l'export afin de pouvoir exporter notre composant, en bas vous ajouter export default et le nom de votre composant donc MyCars.

On a donc importé react et component on a créé une classe qui va utiliser la methode render qui va nous retourner les éléments du return puis on a exporté notre composant. Si on regarde notre app il n'y a aucun changement, pourquoi ?

Et oui nous devons donc l'importer dans notre app donc dans app.js, on y va et on l'importe, regardons le résultat toujours rien, c'est normal on l'importe mais pas encore utilisé et bien faisons le.

```
class App extends Component {
  render() {
    return (
      <div className="App">
        <Mycars />
      </div>
    );
  }
}
```

Pourquoi la majuscule au début et bien imaginons qu'on ait appelé notre composant div par exemple et bien il ne le prendra pas pour un composant mais bien pour la balise div, pour éviter toutes erreurs on écrit donc la première lettre en majuscule. On enregistre et on va voir le résultat et vous pouvez regarder l'inspecteur et vérifier le résultat cela fonctionne enfin youhou. Vous avez donc appris à créer un composant de type classe, à l'importer et l'exporter et aussi l'utiliser. Bravo. Regardons un petit truc ensemble le composant myCars dans votre app vous avez remarqué que cela ressemble beaucoup à du html et bien on peut le transformer regarder

```
render() {
  return (
    <div className="App">
      <Mycars></Mycars>
    </div>
  );
}
```

On pourra éventuellement lui donner des composants enfants par exemple. Mais gardons la première version, regardons ce qui nous reste à faire donc il nous reste le composants Cars.js on va donc le créer mais nous allons faire une fonction cette fois ci. On va dans src et on crée Cars.js dans notre dossier components.



- ⇒ Aide : faire avec une fonction fléchée !!!
- ⇒ Dupliquer une ligne sur vscode shift+alt+flèche du bas

## Exercice 1

Correction :

```
import React from 'react';

const Car = () => {
  return (
    <div>
      <p>Marque: </p>
      <p>Couleur: </p>
    </div>
  )
}

export default Car;
```

Pour l'instant on n'a pas de données mais on va gérer ça avec les props. Dans myCars on a bien importé le composant et afficher cela. Allons voir le resultat.

```
import React, { Component } from 'react';
import Car from './Cars';

class Mycars extends Component {
  render() {
    return (
      <div>
        <h1>Hello React</h1>
        <Car />
      </div>
    )
  }
}

export default Mycars;
```

Donnons un peu de style à cela, ajoutons une classe a la div du composant Car, pour le moment je vous autorise à faire du inline css mais après on bannira cela.

```
<div style={{ backgroundColor: 'pink', width: '400px' }}>
  <p>Marque: </p>
  <p>Couleur: </p>
</div>
```

En fait on crée un objet jsx pour définir le style, on ajoute un padding de 10px et une margin de 5px et auto.

Ensuite on va dupliquer notre composant dans composant MyCars.js de manière à l'avoir 3 fois affiché.

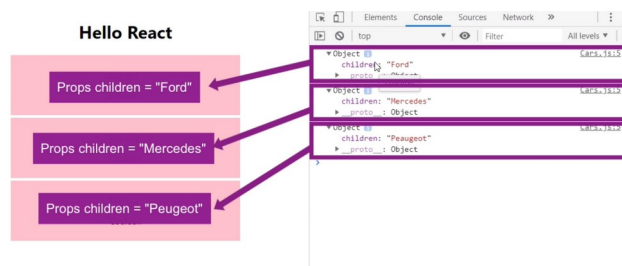
```
class Mycars extends Component {
  render() {
    return (
      <div>
        <h1>Hello React</h1>
        <Car />
        <Car />
        <Car />
      </div>
    )
  }
}
```

Il n'y a aucun intérêt à faire cela comme ça mais dans le cours on va avoir des données en dur donc ça va, imaginons que l'on fasse cela est ce que ça marchera ?

```
<Car>Ford</Car>
<Car>Mercedes</Car>
<Car>Peugeot</Car>
```

Et bien non !! eheh, comme on est dans le composant MyCars et que les infos sont dans un composant enfant on utilisera les **props**. Pour pouvoir récupérer les props via une fonction car là on utilise une fonction il suffit de nous rendre dans le composant Cars.js et ajouter les props en paramètres de notre fonction

```
const Car = (props) => {
  console.log(props);
}
```



Donc pour afficher les marques il faut afficher le children, d'où l'importance du console.log

```
const Car = (props) => {
  console.log(props.children);
}
```

Ça retourne bien la valeur que l'on veut avoir, il suffit donc de les afficher là où on en a besoin. On va devoir afficher du javascript à l'intérieur du jsx on va devoir ouvrir des accolades et demander l'affichage.

```
<div style={ {backgroundColor: 'pink',
  <p>Marque: { props.children } </p>
  <p>Couleur: </p>
}</div>
```

On peut enlever le console.log désormais et on vérifie le résultat. On doit désormais afficher la couleur, là on va la rajouter en paramètre et l'afficher, en vérifiant avec la console log que l'on a bien ajouté cela à l'objet.

```
<Car color="red">Ford</Car>
<Car color="black">Mercedes</Car>
<Car color="green">Peugeot</Car>
```

Ça c'est une méthode de faire mais on a pu voir que l'on avait des objets alors pourquoi ne pas passer les objet en paramètre de la fonction, On va utiliser le destructuring pas de soucie on verra ça en détail plus tard.

Dans notre exemple l'objet contient deux propriétés {children, color}, le destructuring permet d'obtenir ces deux propriétés comme ceci

Const {children, color}=props ;

```
const Car = ({children, color}) => {  
    
  return (  
    <div style={ {backgroundColor: 'pink'} }>  
      <p>Marque: { children } </p>  
      <p>Couleur: {color}</p>  
    </div>  
  )  
}
```

C'est beaucoup plus simple à lire et à afficher. C'est au choix mais je vous conseille le plus simple à relire.

Maintenant imaginons qu'une des voitures n'ai pas de couleur, ce ne sera pas très jolie en visuel on pourra donc remédier à cela en créant une condition. Faisons cela ensemble, au-dessus du return on va donc écrire

const colorInfo = color ? (ici on colle notre <p>couleur) : (<p>Couleur : néant</p>)

ce qui veut dire si on a l'information color alors on affiche qlq chose et sinon on affiche autre chose(les ternaires enfin) il faut désormais utiliser notre condition d'affichage.(true ou false)

```
return (  
  <div style={ {backgroundColor: 'pink'} }>  
    <p>Marque: { children } </p>  
    { colorInfo }  
  </div>  
)
```

On peut même faire encore plus court si vous le désirez

```
<p>Marque: { children }</p>  
{ color ? <p>Couleur: {color}</p> : <p>Couleur: Néant</p> }  
</div>
```

Imaginons qu'il manque désormais une autre info tel qu'une marque comment feriez-vous ?

Encore une fois deux méthodes possible les voici :

```
const Car = ({children, color}) => {

  const colorInfo = color ? (<p>Couleur: { color }</p>) : (<p>Couleur: Néant</p>);

  if(children) {
    return (
      <div style={ {backgroundColor: 'pink', width: '400px', padding: '10px', ma
        <p>Marque: { children } </p>
        { colorInfo }
      </div>
    )
  } else {
    return <p>Pas de data!</p>
  }
}
```

Un if else classique, et maintenant si nous voulions qu'un seul return

```
const Car = ({children, color }) => {

  const colorInfo = color ? <p>Couleur: { color ? color :

  return children ? (
    <div>
      <p>Marque: { children }</p>
      { colorInfo }
    </div>
  ) : <p>Pas de data</p>
}
```

Un ternaire sur le return directement. Si vous n'avez pas de style pensez au style sur la boîte parent donc la div.

Une bonne avancé sur notre appli, mettons désormais le catalogue voiture, on pourra le mettre en dur dans le H1 mais on va faire bcp mieux en abordant le **state**. On va donc le coder ensemble, le state c'est un objet qui va contenir des infos. Ça va se passer sur app.js

```
class App extends Component {

  state = {
    titre: 'Mon catalogue Voitures'
  }
}
```

Il faudrait passer le titre en props au composant MyCars car nous l'avons importé, donc je passe le props ici pour pouvoir le récupérer sur MyCars.js. Je vous montre ça.

```

class App extends Component {
  state = {
    titre: 'Mon catalogue Voitures'
  }

  render() {
    return (
      <div className="App">
        <Mycars title={this.state.titre}/>
      </div>
    );
  }
}

```

Faisons un console.log dans MyCars pour vérifier si on a bien passé le props avec les infos supplémentaire.

```

class Mycars extends Component {
  render() {
    console.log(this);
    return (

```

Pourquoi un this au lieu du props ?

Car on est dans une classe, c'est donc la super variable this qui est utilisé, on l'affichera comme cela.

```

class Mycars extends Component {
  render() {
    console.log(this);
    return (
      <div>
        <h1>{this.props.title}</h1>

        <Car color="red">Ford</Car>
        <Car>Mercedes</Car>
        <Car color="green"></Car>

      </div>
    );
  }
}

```

Le titre vient désormais du state, si lui est modifié le titre changera. On a bien avancé, on a vu les composants state et sans state (state : construit via une classe) (sans-state : construit via une fonction fléchée), comment récupérer les props et les utiliser et un petit exercice.

## Exercice 2

```
class App extends Component {  
  state = {  
    titre: 'Mon Catalogue Voitures 2',  
    color: 'green'  
  }  
  
  render() {  
    return (  
      <div className='App'>  
        <Mycars  
          color={this.state.color}  
          title={this.state.titre}  
        />  
      </div>  
    )  
  }  
}
```

Création du state color et ajout de propriété aux composants.

On va dans le composant MyCars.js et on ajoute le tout.

```
class Mycars extends Component {  
  
  render() {  
    return (  
      <div>  
        <h1 style={{ color: this.props.color }}>{this.props.title}</h1>  
  
        <Car color="red">Ford</Car>  
        <Car>Mercedes</Car>  
        <Car color="green"></Car>  
      </div>  
    )  
  }  
}
```

On save et on teste, passons à la suite, on crée le state sur MyCars.js,

```
class Mycars extends Component {

  state = {
    cars: ["Ford", "mercedes", "Peugeot"]
  }
}
```

Et ensuite on appelle nos infos de manière à les afficher.

```
render() {
  return (
    <div>
      <h1 style={{ color: this.props.color }}>{this.props.title}</h1>

      <Car color="red">{this.state.cars[0]}</Car>
      <Car>{this.state.cars[1]}</Car>
      <Car color="green">{this.state.cars[2]}</Car>
    </div>
  )
}
```

Voilà pour les props et pour les state. Beau travail jusqu'à présent.

Imaginons que l'on ait besoin de répéter un titre un certain nombre de fois dans plusieurs composants, on pourrait l'extraire dans un composant à part, faisons cela.

Dans components vous créez un fichier MyHeader.js sur lequel on va partir sur une fonction fléchée, (arrow function) on peut enlever ou garder le return dans ce cas (retourne 1 seul élément)

```
const MyHeader = () =>
  <h1></h1>

export default MyHeader
```

Ainsi qu'un fichier Wrapper.js sur lequel on fera une fonction,

```
function Wrapper() {
  return (
    <div style={{ backgroundColor: 'pink', width: '400px', padding: '10px', margin: '5px auto' }}>
      .....
    </div>
  )
}

export default Wrapper
```



On importe notre wrapper dans le composant Cars.js et on l'utilise en tant qu'élément parent des deux paragraphes(<p>),

```
return children && (  
  <Wrapper>  
    <p>Marque: { children }</p>  
    <p>Couleur: {colorInfo}</p>  
  </Wrapper>  
)
```

À la suite de ça nous allons extraire cela dans le Wrapper.js, en paramètre de la fonction et évidemment afficher le rendu dans notre div.

```
function Wrapper({ children }) {  
  return (  
    <div style={{backgroundColor: 'pink', width: '400px', padding: '10px', margin: '5px auto'}}>  
      {children}  
    </div>  
  )  
}  
  
export default Wrapper
```

En fait nous avons injecté la div directement au niveau du dom, on rafraîchit et elle va disparaître au niveau du header, on peut le voir dans l'inspecteur. Faisons la même chose sur le composant MyCars.js

```
return (  
  <div>  
    <Wrapper>  
      <h1 style={{color: this.props.color}}>{this.props.title}</h1>  
    </Wrapper>  
  
    <Car color="red">{this.state.cars[0]}</Car>  
    <Car>{this.state.cars[1]}</Car>  
    <Car color="green">{this.state.cars[2]}</Car>  
  </div>  
)
```

Save et regarder le résultat la div et la couleur reviennent au niveau du H1, on peut vérifier avec l'inspecteur. Vous pouvez faire un console.log du children dans le wrapper et vérifier dans votre console, vous verrez les 4 objets. Tout ça pour comprendre que tout composant peut contenir des composants qui eux même peuvent contenir du html ou des composants. Finissons nos H1, sur MyCars.js nous allons importer le composant MyHeader.js et gérer le style ailleurs.

```

return
    <div>
      <Wrapper>
        <MyHeader>
          {this.props.title}
        </MyHeader>
      </Wrapper>

      <Car color="red">{this.state.cars[0]}</Car>
      <Car>{this.state.cars[1]}</Car>
      <Car color="green">{this.state.cars[2]}</Car>
    </div>

```

Sur MyHeader.js et bien j'ajoute le style

```

const MyHeader = () => <h1 style={{ color: this.props.color}}></h1>

export default MyHeader

```

Sauf que ici dans notre objet on a bien color mais this.props.color lui on ne la pas ici on la dans MyCars.js, Comment le recupere t-on ??

En le passant en props du composant MyHeader dans le fichier MyCars.js exactement.

```

<Wrapper>
  <MyHeader
    myStyle={this.props.color}
  >
    {this.props.title}
  </MyHeader>
</Wrapper>

```

On le passe donc via le destructuring au composant MyHeader,

```

const MyHeader = ({ myStyle }) => <h1 style={{ color: myStyle}}></h1>

export default MyHeader

```

Sauf que ce n'est pas totalement terminé, on a importé le style mais il reste encore l'élément enfant, et enfant on l'appelle avec quel paramètre ???

Children bravo !!!

```

const MyHeader = ({ myStyle, children }) => <h1 style={{ color: myStyle}}>{children}</h1>

export default MyHeader

```

De ce fait, partout où j'importe mon composant je n'aurais qu'à lui passer ces deux informations.

Imaginons maintenant que je veuille mettre un deuxième titre plus bas et bien je n'ai plus qu'à coller le composant.

```
<Wrapper>
  <MyHeader
    myStyle={this.props.color}
  >
    {this.props.title}
  </MyHeader>
</Wrapper>

<Car color="red">{this.state.cars[0]}</Car>
<Car>{this.state.cars[1]}</Car>
<Car color="green">{this.state.cars[2]}</Car>

<MyHeader
  myStyle={this.props.color}
>
  Bonjour
</MyHeader>
```

Je peux lui passer une chaîne de caractère pour changer le titre évidemment et le style quand à lui reste le même à moins de le définir.

De cette manière-là, vous pouvez créer des composants que vous allez utiliser partout dans votre application.

Nous allons maintenant mettre deux composants bien distincts dans un seul fichier afin de les regrouper, imaginons que l'on ait besoin uniquement de ces deux composants dans une section bien spécifique de notre appli dans ce cas il ne sera pas nécessaire de créer deux fichiers. (Cars et MyCars) je conseille vivement de les séparer mais nous allons voir comment effectuer cela. Sur Cars.js nous allons prendre toute la arrow Function sans l'export, et sur MyCars.js nous redéfinissons notre arrow Function relative aux composants car celle-ci nous retourne cet élément jsx et via Car c'est-à-dire le composant lui-même sera utilisé au niveau de la classe, c'est à dire au niveau de notre jsx.

```
<Car color="red">{this.state.cars[0]}</Car>
<Car>{this.state.cars[1]}</Car>
<Car color="green">{this.state.cars[2]}</Car>
```

```
const Car = ({children, color}) => {
  const colorInfo = color ? color : "Néant";

  return children && (
    <Wrapper>
      <p>Marque: {children}</p>
      <p>Couleur: {colorInfo}</p>
    </Wrapper>
  )
}

export class Mycars extends Component {
```

Point important, dans React les props sont immuables c'est-à-dire que vous n'êtes pas censé les modifier, cependant dans certaines situations vous pourrez être amené à vouloir les modifier, dans ce cas la imaginons si je prends children (dans Wrapper.js) on devra l'extraire via le destructuring vous pouvez essayer si vous le voulez.

```
function Wrapper({ children }) {
  children = 'Bonjour';

  return (
    <div style={{ backgroundColor:
      { children }
    </div>
  )
}
```

Cela va bien changer tous les children en « bonjour » mais nous l'avons écrit en dur au lieu de récupérer les infos, on évite vraiment de faire cela, pensez à la logique interne de react qui consiste à passer les props pour pouvoir les réutiliser dans les composant enfant. On pourra le faire directement sur props mais vous allez voir react ne va pas aimer

```
function Wrapper(props) {
  props.children = 'Bonjour';
  return (
    <div style={{ backgroundColor: '
      { props.children }
    </div>
  )
}
```

Allez tester votre appli, il n'y a rien et un message d'erreur est apparu dans la console, donc on garde à l'esprit que les props sont immuables.

Nous allons aborder quelques événements, petit à petit, et toujours en pratiquant bien évidemment. On va commencer par le `onCopy`, il est rarement utilisé mais c'est un des plus simple donc commençons par celui-ci.

On va faire en sorte de faire un petit paragraphe sous notre titre de catalogue et si quelqu'un essaie de le copier on lui affichera un pop-up.

Allons là où se situe notre titre, donc au niveau de `MyCars.js` juste sous le titre `H1`, créons un `p` avec `lorem ipsum` pour le texte, on enregistre et on vérifie. Pour créer un alert il faudrait faire quoi ???

Une fonction oui c'est bien cela, donc sous notre classe, on vient créer notre arrow function,

```
class Mycars extends Component {  
  
  noCopy = () => {  
    alert('merci de ne pas copier le texte');  
  }  
  
  render() {
```

Qui va se lancer contre qlq'un qui essaie de copier notre texte, on va devoir créer l'évènement sur cet élément précis.

```
<p onCopy={this.noCopy}>Lorem ipsum,
```

Enregistrez et essayez de copier ce texte, une pop up doit apparaître. C'était comme je vous l'ai dit très simple. Essayons désormais le `onMouseOver`, (très peu utilisé également) disons que lorsque l'on passe sur notre titre il passe en majuscule et change de couleur. Pour cela on va faire du CSS mais pas en inline style, rendons-nous sur `index.css` et modifions cela.

```
h1.styled {  
  color: red;  
  text-transform: uppercase;  
}
```

J'ai créé une classe, et si nous avons cette classe alors l'élément aura les propriétés

Nous allons injecter cette classe si qlq'un passe la souris dessus, créons la fonction et ajoutons là.

```

addStyle = (e) => {
  console.log(e.target);

  if(e.target.classList.contains('styled')) {
    e.target.classList.remove('styled');
  } else {
    e.target.classList.add('styled');
  }
}

```

Ici nous avons écrit que si on passe sur l'élément et qu'il a déjà la classe on l'enlève et sinon on l'ajoute tout simplement.

```

render() {
  return(
    <div>
      <h1 onMouseOver={this.addStyle}>{this.props.title}</h1>
    </div>
  );
}

```

(PENSER A PRECISER QUE L'ON REPETE LE H1 MAIS QUE C'EST LE MEME CE N'EST PAS UN DEUXIEME H1, L'UN LE CREE ET L'AUTRE ON AGIT DESSUS)

Allons vérifier cela, en ouvrant la console comme cela on voit bien ce qu'il se passe en direct, voilà vous avez appris à travailler sur les événements.

Revenons sur App.js et continuons à avancer, on va voir le onClick, on va créer un bouton et modifier le titre par exemple au click, on reste sur des données en dur et ensuite on avancera. Commençons par créer un bouton et on va penser à commenter ou enlever le onMouseOver pour que cela ne nous perturbe pas.

```

render() {
  return (
    <div className="App">
      <Mycars title={this.state.titre}/>
      <button>Changer le nom en dur</button>
    </div>
  );
}

```

On va faire en sorte qu'au click sur ce bouton le contenu se modifie

### Exercice 3

```

changeTitle = (e) => {
  this.setState({
    titre: 'Mon nouveau titre'
  })
}

```



On a ici une arrow function qui change le titre une fois que l'on a click sur le button, la fonction va appeler la méthode setState qui elle a le droit de modifier le state(donc le titre)

```
<button onClick={this.changeTitle}>Changer le nom en dur</button>
```

On a bien réussi à changer le titre avec des données en dur, on va maintenant créer une fonction dans laquelle on va passer un paramètre et c'est ce dernier qui va mettre à jour le contenu en question.

```
changeViaParam = (titre) => {  
  this.setState({  
    titre: titre  
  })  
}
```

Créons un deuxième bouton, et appliquons cela,

```
<button onClick={this.changeTitle}>Changer le nom en dur</button>  
<button onClick={() => this.changeViaParam('Titre via un parm')}>Changer le nom
```

Ici nous avons notre fonction qui a un paramètre, qui une fois lancée, elle va prendre le paramètre en question, nous avons ajouté une fonction fléchée pour ne pas que la fonction se lance à chaque chargement de la page, juste au click. Allez testons cela.

Ça fonctionne et bien je vous propose une méthode bien meilleure, on va le faire par un **Bind**, je vous invite à aller voir ce qu'est le bind sur la docs. Alors faisons cela ensemble :

Créons une fonction, qui va récupérer le param grâce à la fonction fléchée

```
changeViaBind = (param) => {  
  this.setState({  
    titre: param  
  })  
}
```

Nous allons désormais faire la modification sur notre button, en utilisant la méthode bind, puis on va définir notre titre.

```
<button onClick={() => this.changeViaParam('Titre via un parm')}>Via param</button>  
<button onClick={this.changeViaBind.bind(this, 'Titre via Bind')}>Via Bind</button>
```

Vérifions le résultat, et faisons un rappel grâce à la fonction changeViaBind on va récupérer le state et le modifier grâce au param cela va le mettre à jour. Là on est sûr des données en dur alors oui cela n'a pas vraiment de sens de faire cela mais sur un formulaire par exemple cela peut-être très intéressant. Alors on ne va pas créer un formulaire mais créons un input ça va être une donnée manipulée par l'utilisateur qui va faire en sorte de modifier le contenu de la page.



```
<button onClick={this.changeViaBind.bind(this, 'Titre via Bind')}>Via Bind</button>  
<input type="text" onChange={this.changeViaInput} value={this.state.titre}/>
```

Créons maintenant la fonction, avec la méthode onChange et la valeur qui sera le titre par défaut. Pour le moment on ne peut pas le modifier et on régler cela. Il va falloir que l'on récupère les nouvelles données et mettre à jour le state.

```
changeViaInput = (e) => {  
  this.setState({  
    titre: e.target.value  
  })  
}
```

On met à jour avec le setState qui lui va prendre la valeur que l'on a récupéré via le eventObject (e, comme on est dans une classe), pour mettre à jour cette info. Mettons à jour et vérifions, on peut modifier et cela se met à jour automatiquement.

### PETIT RAPPEL

#### Note sur la mutation du state et quelques mauvaises pratiques à éviter.

Dans la partie précédente, je vous ai présenté rapidement le constructor(props) qui n'est plus obligatoire depuis React 16.

En effet, si vous n'initialisez pas d'état local et ne liez pas de méthodes, vous n'avez pas besoin d'implémenter votre propre constructeur pour votre composant React. Vous définissez directement votre state sans le constructeur. Donc, sans le this.

Je vous ai également dit que "***vous ne devez jamais modifier le state directement ! Vous devez toujours passer par la méthode setState()***". Ça, c'est la procédure courante à l'extérieur du constructeur.

Cependant, si vous avez besoin d'initialiser l'état local de votre composant en affectant un objet à this.state ou si vous souhaitez lier des méthodes gestionnaires d'événements à l'instance vous allez donc devoir implémenter le constructeur. Dans ce cas-là, la procédure est un peu différente.

**Exemple 1:** Vous avez besoin d'initialiser l'état local de votre composant en affectant votre state (l'objet this.state).

Dans ce cas, vous ne devez **SURTOUT PAS** appeler setState() dans le constructor()! Au lieu de ça, affectez directement l'état initial à this.state dans le constructeur.

1. constructor(props) {
2.   super(props);
- 3.

```
4.    // N'appellez pas `this.setState()` ici !  
5.    this.state = { counter: 0 };  
6. }
```

**Exemple 2:** Si vous souhaitez lier des méthodes gestionnaires d'événements à l'instance, vous allez donc devoir implémenter le constructeur.

```
1. constructor(props) {  
2.     super(props);  
3.     this.handleClick = this.handleClick.bind(this);  
4. }
```

**Gardez donc bien cette information en tête:**

Le constructeur est le seul endroit où vous devriez affecter directement une valeur à `this.state` sans passer par la méthode `setState()` car cette dernière est strictement interdite dans le `constructor()`.

Une autre erreur courante chez les débutants en React, consiste à copier les props dans l'état local. Ne faites jamais ça !

**Exemple:**

```
1. constructor(props) {  
2.     super(props);  
3.  
4.     // Ne faites pas ça !  
5.     this.state = { color: props.color };  
6. }
```

Comme nous avons altéré légèrement les fichiers de notre application dans les exercices précédents, je vous invite à télécharger le fichier nommé "3-fichiers.zip".

Une fois téléchargé, dézippez-le pour extraire les trois fichiers "Car.js", "MyCar.js" et "App.js" et utilisez-les pour remplacer les fichiers du même nom dans votre application "cars". Ainsi, vous aurez exactement le même code que moi pour pouvoir suivre la suite du cours.

Pour ceux qui veulent garder leur code je vous invite à le push ou le garder sur votre ordi.

On va retourner sur notre appli, nous allons ajouter une donnée comme l'année du véhicule. Allons sur MyCars.js,

```
<Car year={"2000"} color="red">Ford</Car>
<Car>Mercedes</Car>
<Car color="green"></Car>
```

Nous avons plus qu'à rappeler le props, pour l'affichage.

```
const Car = ({color, children, year}) => {
```

Et également dans notre p,

```
<p>Marque: {children}</p>
<p>Année: {year}</p>
{ colorInfo }
```

Allons vérifier, mais nous on ne va pas faire comme cela eheh, ce que je vous propose c'est de créer un state et on va mettre dedans les différentes voiture de notre catalogue.

**Petit Rappel** le mot **state** est un mot réservé on ne peut pas l'utiliser ailleurs, il sert pour la notion de state.

Nous allons créer un objet tableau dans le state.

#### Exercice 4 :

```
state = {
  voitures: [
    {name: 'Ford', color: 'red', year: 2000},
    {name: 'Mercedes', color: 'black', year: 2010},
    {name: 'Peugeot', color: 'green', year: 2018},
  ]
}
```

Allons désormais récupérer les infos et les afficher.

```
<Car color={this.state.voitures[0].color}>{this.state.voitures[0].name}</Car>
<Car>{this.state.voitures[1].name}</Car>
<Car>{this.state.voitures[2].name}</Car>
```

Evidemment pour afficher toutes les infos nous devons les rappeler. Bien joué si vous avez réussi. **Recap**, on a notre composant Car, on a créé un state qui contient les infos à afficher, on a appelé les props et on a afficher nos données. On va juste faire une légère modification je vous propose de mettre l'âge du véhicule plutôt que

l'année. Comment faire et bien on va prendre l'année actuelle et on va déduire la date de construction du véhicule.

Créons un bouton simple, pour dire que l'on peut vieillir un véhicule par exemple.

```
<button> + 10 ans</button>
<Car color={this.state.voitures
```

Créons désormais la fonction qui va gérer cela, on en est plus à notre première maintenant. Ici on va utiliser la méthode map ( qui va nous permettre d'aller chercher chaque élément dans le tableau et en faire ce que l'on veut) pour ceux qui veulent en savoir plus je vous invite aller voir la docs pour cette méthode. Voyons ensemble sur w3school Javascript map().

```
addTenYears = () => {
  const updatedState = this.state.voitures.map((param) => {
    return param.year -= 10;
  })
  this.setState({
    updatedState
  })
}
```

Ici, j'utilise l'opérateur d'affectation après soustraction ( -= ) pour me calculer la soustraction de l'opérande gauche par l'opérande droit puis affecter le résultat à la variable représentée par l'opérande gauche.

Il faut désormais gérer l'affichage. Créons une constante qui nous renverra l'année en cours.

```
const year = new Date().getFullYear();
return(
  <div>
```

La méthode getFullYear() renvoie l'année de la date renseignée d'après l'heure locale. Ainsi, j'aurais l'année en cours dans "year"

Et enfin l'affichage avec la logique

```
{year - this.state.voitures[0].year + 'ans'}>
{year - this.state.voitures[1].year + 'ans'}>
{year - this.state.voitures[2].year + 'ans'}>
```

Comme nous sommes dans un contexte JS entre les {} on peut donc parfaitement effectuer des opérations de logique et autres calculs via la JS. Plus tard on verra comment le gérer avec une fonction. Allons voir notre rendu.

On est d'accord que là nous avons un code assez long et compliqué alors je vous propose de faire mieux que cela,

```

<Car color={this.state.voitures[0].color} year
<Car color={this.state.voitures[1].color} year
<Car color={this.state.voitures[2].color} year

{
  this.state.voitures.map(voiture => {
    return(
      <Car color={voiture.color}></Car>
    )
  })
}

```

On va donc utiliser la methode map sur l'objet voiture, donc plus besoin de préciser l'index, et on ajoute la couleur et l'année,

```

<Car color={voiture.color} year={year - voiture.year + 'ans'}>{voiture.name}</Car>

```

On pourra avoir 100, 1000 voitures ou plus et bien là on a réglé problème, on peut alors effacer tout ce code inutile.

```

<button onClick={this.addTenYears}> + 10 ans</button>

{
  this.state.voitures.map(voiture => {
    return(
      <Car color={voiture.color} year={year - voiture.year + 'ans'}>{
    }
  })
}

```

Enregistrons mais je vous préviens nous allons avoir une erreur, pourquoi ??? Regardez votre console.

Et bien en fait lorsqu'on utilise map pour afficher à ce niveau React aimerait avoir une **clé(key)** qui va faire en sorte qu'il puisse récupérer ou identifier chaque élément, un peu comme un **id**. Imaginons qu'on veuille supprimer un modèle et bien il ne saura pas lequel donc on va lui préciser en ajoutant un paramètre (un index) et en modifiant légèrement notre code.

```

this.state.voitures.map((voiture, index) => {
  return(
    <Car key={index} color={voiture.color}
  )
})

```

On peut également le faire d'une autre manière, de manière plus propre dirons-nous je vous montre ça de suite.

```

return(
  <div key={index}>
    <Car color={voiture.color} year={year - voiture.year + 'ans'}>
  </div>
)

```

Comme cela il va créer un bloc avec l'index que nous désirons et à chaque modèle il créera un nouveau bloc(div) avec chaque élément. Ici nous avons utilisé le children mais on pourra encore une fois modifier si besoin, là cela fonctionne donc tout va bien.

Mais profitons-en pour revoir le destructuring et optimiser encore un peu. Eh oui je vous martyrise. Passez votre souris sur l'objet voiture qui est en paramètre et vous allez voir ce qu'il contient, et on va pouvoir le déstructurer afin de faciliter l'écriture du code.

```
this.state.voitures.map(({name, color, year}, index) => {
```

Maintenant on passe à l'affichage comme toujours :

```
return (  
  <div key={index}>  
    <Car  
      nom={name}  
      color={color}  
      year={year} />  
    </div>  
  )
```

Ça va fonctionner super bien. Ajoutons un autre détail, imaginons que notre voiture soit sortie l'an dernier on voudrait qu'il y ait écrit 1 an sans le s, on est d'accord, alors changeons cela.

```
getAge = year => {  
  const now = new Date().getFullYear();  
  const age = now - year;  
  
  // ans, an,  
  let frenchYearStr = ""  
  if (age === 1) {  
    frenchYearStr = "an";  
  } else if (age > 1) {  
    frenchYearStr = "ans";  
  }  
  return `${age} ${frenchYearStr}`;  
}
```

Les littéraux de gabarits sont des littéraux de chaînes de caractères permettant d'intégrer des expressions. Avec eux, on peut utiliser des chaînes de caractères multi-lignes et des fonctionnalités d'interpolation.

**Attention! Pour les définir, vous devez utiliser l'accent grave ` au lieu des apostrophes simples ' '**

Si vous voulez gérez le cas de l'année +0 a vous de jouer et rajouter la condition.

## Exercice 5

### Correction exercice 5

```
const Welcome = () => {  
  
  const bonjour = () => console.log('Bonjour');  
  
  return (  
    <div>  
      /* Invoker une fonction "bonjour" qui affichera console.log('Bonjour') */  
      <button onClick={bonjour}>Invoker une fonction</button>  
    </div>  
  );  
}
```

Suite

```
const bonjour = () => console.log('Bonjour 2');  
const bonsoir = arg => console.log(arg);
```

On crée la fonction puis on passe l'argument pour affichage. Comme on a un argument on passe en arrow function.

```
/* Invoker une fonction "bonsoir" avec un argument "Bonsoir"  
et l'afficher dans un console.log  
*/  
<button onClick={() => bonsoir("Bonsoir")}>Invoker une fonction avec arg "Bonsoir"</button>
```

Le dernier sans invocation de fonction

```
/* lancer le console.log("Bonne nuit") après le click sans invoquer de fonction */  
<button onClick={() => console.log("Bonne nuit")}>Clg sur le bouton</button>
```

Bravo à tous si vous avez réussi vous avez donc bien compris les différences, alors bravo c'est un gros pas en avant là. Ceci pour vous rappeler qu'entre les accolades on peut mettre du JS.

Alors jusqu'ici on a appris à créer nos props, les récupérer au niveau du composant enfant, nous avons appris à passer de la data, de la récupérer par le state, ou bien de créer de la data en dur pour tel ou tel composant, alors essayons d'aller plus loin désormais.

Nous allons passer de la data par une fonction, on va passer une fonction en props et la faire exécuter au niveau de l'élément parent. **Explication** : l'évènement lui va venir de l'élément enfant, c'est-à-dire de la data qui vient d'en bas vers le haut.

Donc pour illustrer la suite on va partir sur la notion de parent enfant comme dans la vraie vie, pour cette exercice nous allons créer une nouvelle app, cela vous fera le plus grand bien. Allons sur le App.js et débarrassons-nous du superflu comme toujours.

Allons créer notre dossier components qui contiendra nos deux enfants, Maman.js et Toto.js. Importons React et {component} puis créons la classe qui étend Component. Ensuite nous aurons quoi ????

Un state qui n'est pas obligatoire mais pour l'exercice oui, et la méthode render qui retournera une div, nous aurons la base. A vous de jouer.



```
import React, { Component } from 'react';

class Maman extends Component {

  state = {

  }

  render() {
    return (
      <div>

      </div>
    )
  }
}
```

On va dire que dans notre state on va avoir deux messages, un qui est la maman qui dit a son fils d'aller ranger sa chambre et celui de son fils qui dit qu'il est d'accord, il seront donc tous les deux dans leur composant parent qui est Maman. Pour le moment on va les mettre en null.

```
state = {
  messageMaman: null,
  messageToto: null
}
```

Une fois que cela est fait on va devoir exporter Maman, je vous conseille de faire l'export directement quand on crée la structure cela évite des oublis. Importons le dans le App.js puis affichons notre composant Maman.

```
import React from 'react';
import Maman from './components/Maman';
import './App.css';

function App() {
  return (
    <div className="App">
      <Maman />
    </div>
  );
}

export default App;
```

Créons la partie Maman et mettons une ligne en dessous,

```
render() {
  return (
    <div>
      <h1>Maman</h1>

      <hr />
    </div>
  )
}
```

Pour simuler l'ordre de la maman nous allons créer un bouton 'ordre de la maman', et on va créer un 'p' qui affichera le dit message. On a vu précédemment comment créer un évènement sur un bouton et exécuter une fonction, alors faisons ça.

```
ordreMaman = () => {  
  
}  
  
render() {  
  return (  
    <div>  
      <h1>Maman</h1>  
      <button onClick={this.ordreMaman}>Ordre de la mère</button>  
    </div>  
  );  
}
```

Créez la fonction, c'est à vous.

```
ordreMaman = () => {  
  this.setState({  
    messageMaman: 'Va ranger ta chambre'  
  })  
}
```

Il se passe quoi désormais ??? Rien et bien c'est normal il faut gérer l'affichage, alors allez-y.

```
<h1>Maman</h1>  
<button onClick={this.ordreMaman}>Ordre de la mère</button>  
<p>{this.state.messageMaman}</p>  
<hr />
```

**Petit rappel** de ce qui se passe, au click sur le bouton, on active la fonction qui met à jour le state du messageMaman et nous allons le récupérer avec notre message. Vérifiez que cela fonctionne. Très bien passons au composant Toto, pour qui nous allons créer une fonction. A vous de jouer.

```
import React from 'react';  
  
const Toto = () => {  
  
  return(  
    <div>  
      je suis Toto  
    </div>  
  );  
}  
  
export default Toto;
```

Importons Toto dans son parent donc ??? Et oui Maman, `import Toto from './Toto';` et une fois importé on va devoir l'afficher.

On va désormais faire qlq chose qui ressemble au composant Maman,

```

<h1>Maman</h1>
<button onClick={this.ordreMaman}>Ordre de la mère</button>
<p>{this.state.messageMaman}</p>
<hr />
<Toto name="Toto" />

```

On va écrire le name de cette manière afin qu'il ne soit pas écrit en dur, mais plutôt en props.

```

const Toto = props => {
  return(
    <div>
      <h2>{props.name}</h2>
    </div>
  )
}

```

On va créer également un bouton mais qui ne sera affiché que si Maman lui donne l'ordre d'aller ranger sa chambre, comment faire cela ???

Avec une condition exactement. Il s'affichera que lorsque le state sera mis à jour avec l'ordre de Maman.

```

<Toto name="Toto" leState={this.state}/>

```

Je vous encourage à faire un console.log de props dans le composant Toto afin de vérifier que l'on est dans la bonne direction. Si oui alors on peut commencer notre condition,

```

const Toto = props => {
  return(
    props.leState.messageMaman !== null ? (<button>Réponse</button>) : (<button disabled>Réponse</button>)
  )
}

```

Et nous allons mettre cela dans une constante,

```

const btnReponseToto = props.leState.messageMaman

```

**Petit Récap :** au niveau de maman on a passé le state (this.state) en props = leState, on accède au message maman, si il est différent de null il affiche le bouton afin que toto puisse répondre sinon on affiche le bouton mais il est désactiver. On va afficher tout cela.

```

return(
  <div>
    <h2>{props.name}</h2>
    {btnReponseToto}
  </div>
)

```

On enregistre et on vérifie. Tout fonctionne bien. Maintenant on va faire en sorte que lorsque la maman a donné l'ordre et bien toto lui réponde. Cette fois ci on va le faire dans le composant parent donc maman. Donc **passer une fonction comme props**, sous la fonction ordreMaman on va donc créer la fonction réponseToto afin d'afficher la réponse de toto.

```

reponseToto = () => {
  this.setState({
    messageToto: "D'accord maman"
  })
}

```

On va passer la fonction en props sur le composant enfant à l'intérieur du composant parent (donc toto dans maman) Je sais c'est sale mais passons ce détail 😊, on va devoir récupérer le props via son nom,

```

<Toto name="Toto" reponseToto={this.reponseToto} leState={this.state}/>

```

Ensuite nous allons devoir l'ajouter à notre condition d'affichage évidemment,

```

(<button onClick={props.reponseToto}>Réponse</button>

```

Attention on ne passe pas la fonction mais bien le nom du props dans cet exemple il s'appelle pareil mais on aurait pu lui donner un nom différent, on a plus qu'à gérer l'affichage de la réponse de toto en le mettant dans un 'p' comme pour maman, dans Toto.js bien sûr.

```

return(
  <div>
    <h2>{props.name}</h2>
    {btnReponseToto}
    <p>{props.leState.messageToto}</p>
  </div>
)

```

On enregistre et on test, et hop tout fonctionne bien.

## Exercice 6

### Correction exercice 6

Commençons par ordreMaman sur le bouton on a l'évènement onclick qui invoque notre méthode ordreMaman et qui lui passe en argument la phrase 'Va ranger ta chambre', en amont il y a une arrow function nous devons donc récupérer cette chaîne de caractère au niveau de la fonction.

```

ordreMaman = msg => this.setState({ messageMaman: msg });
reponseToto = msg => this.setState({ messageToto: msg });

```

Première étape faite, ensuite agissons sur le disabled,

```

ordreMaman = msg => this.setState({ messageMaman: msg, disabled: false });
reponseToto = msg => this.setState({ messageToto: msg });

```

Agissons désormais sur Toto, nous allons devoir récupérer le state qui est déjà écrit dans le composant enfant de maman, donc toto on va donc récupérer le this.state,

```
<button
  disabled={props.leState.disabled}
>Réponse</button>
```

Dans le bouton on a géré les deux cas, si la maman a parlé ou pas (true ou false pour le disabled), ensuite nous avons la méthode onclick, que nous avons véhiculé via le props responseToto

```
<Toto
  name="Toto"
  reponseTotoProps={this.reponseToto}
  leState={this.state}
/>
```

On se rend sur Toto.js et on ajoute cela

```
<button
  disabled={props.leState.disabled}
  onClick={props.reponseTotoProps}
>Réponse</button>
```

Dans ce dernier nous avons donc notre fonction responseToto, nous l'avons juste invoqué et dedans il y a un argument msg que l'on à utiliser pour mettre à jour la propriété messageToto que nous avons dans le state, nous allons donc devoir au moment d'invoquer la fonction lui fournir cet argument, pour mettre à jour le state du messageToto.

```
<button
  disabled={props.leState.disabled}
  onClick={() => props.reponseTotoProps("Non, je veux regarder la télé!")}
>Réponse</button>
```

Et voilà l'exercice est terminé, bravo à vous tous.

### Notion de destructuring :

Qu'est-ce que cela veut dire ??? Et bien c'est une des nouveautés apporté à Js dans sa version ES6 et qui nous facilite simplement l'extraction de la data que ce soit dans un array, un tableau ou dans un objet pour les placer dans des variables distinctes, c'est aussi simple que ça.

### Les conditions dans React :

C'est exactement la même chose qu'en Javascript (if, if else etc...)

### Les images en React :

Nous allons voir comment afficher une image jpg, png, ou svg comme le logo React au début de votre appli dans le App.js. Allons chercher une image sur [icone finder](#) chercher les différents formats. Si on va dans l'éditeur on peut par exemple changer la couleur de l'image, on clique download et on a l'image, on l'enregistre dans son fichier c'est plus simple. Je vais l'afficher dans un composant a part pour l'exemple, on importe l'image dans les imports **en précisant son format c'est obligatoire**. Créons deux nouveau fichier un dans le composants qui sera Car.js pour les test image, et un qui sera Form.js.

```
import React from 'react';
import car from './car.png';

const Car = () => {
  return <img src={car} alt="" />
}

export default Car;
```

Ensuite si on veut l'afficher dans un autre composant on fait de cette manière, évidemment on importe notre composant

```
class Form extends Component {
  render() {
    return(
      <div>
        <Car />
        <h1>Commentaire</h1>
        Formulaire
      </div>
    )
  }
}
```

Et hop elle s'affiche pour un autre format c'est exactement pareil, voilà vous savez faire. Si format SVG on peut procéder différemment car le code svg nous donne la width et la couleur etc... Click sur l'image dans notre arborescence pour avoir acces au code svg de l'image.

```
const Car = () => {
  return (
    <svg width="512" height="512" xmlns="http://www.w3.org/2000/svg">
      <g>
        <title>background</title>
        <rect fill="none" id="canvas_background" height="402" width="512" />
      </g>
      <g>
        <title>Layer 1</title>
        <path fill="#222222" fill-rule="nonzero" stroke-width="12.25" />
      </g>
    </svg>
  )
}
```

La balise svg sert de parent donc pas oblige de mettre dans une div. On pourra via le composant (<Car />) dans notre cas lui passer des props pour gérer la couleur, la width etc... On peut aller dans le index.css et créer une classe pour lui mettre par exemple une bordure.

```
.carBorder{
  border: 1px solid red;
}
```

Ensuite on se rend sur notre élément afin de lui attribuer la class en question, (on pourra remplacer la class par svg dans cet exemple mais cela s'appliquera a tous les svg

```
<svg className="carBorder" width="512" height="512">
```

Modifions la couleur et la taille maintenant, en props.

```
<Car color="red" height="400" />
```

Pour récupérer les props comment fait-on ???

En paramètre de la fonction exactement, `Car = (props) => {`

Et on les appelle sur le code de l'image svg,

```
<svg className="carBorder" width="512" height={props.height}
```

```
<path fill={props.color}
```

Vous pouvez enregistrer et tester, cela fonctionne très bien, on peut avoir des erreurs dans la console, c'est très simple à résoudre

```
fillRule="nonzero" strokeWidth="12.257193"
```

C'est très souvent écrit avec un tiret et bien on l'enlève et on écrit en camelCase et l'erreur disparaît. C'est pas plus compliqué que ça.

### Les formulaires en React :

Reprenons sur Form.js, nous allons travailler avec un input type text, un select ainsi qu'un text-area, pensez à importer le composant form dans le App.js, cela devrait vous rappeler qlq chose.

```
return(  
  <div>  
    <Car color="red" height="400" />  
    <h1>Commentaire</h1>  
  
    <form>  
      <label>Pseudo</label>  
      <input type="text" value="" />  
    </form>  
  </div>  
)
```

Créons un state, avec les propriétés que nous voulons avoir donc un username par exemple,

```
class Form extends Component {  
  state = {  
    username: '',  
  }  
}
```

Nous allons bien sûr le récupérer dans l'input,



```

<form>
  <label>Pseudo</label>
  <input type="text" value={this.state.username} />
</form>

```

Pour modifier le contenu du champ il va falloir faire appel à une fonction et une méthode, créons-la.

```

handlePseudo = e => {
}

```

Evidemment nous allons l'appeler dans l'input,

```

<form>
  <label>Pseudo</label>
  <input type="text" value={this.state.username} onChange={this.handlePseudo} />
</form>

```

Ici la fonction s'enclenche de manière répétitive, transfère les données via le value, dans la méthode. Pour cela il faut que l'on récupère les event object ainsi que le target et la value.

```

handlePseudo = e => {
  this.setState({
    username: e.target.value
  })
}

```

Si vous êtes sur chrome vous pouvez télécharger React developers tools en extension sinon on fera sans, (ça permet d'accéder au state des composants React, tout fonctionne passons maintenant au select, mais on va organiser un peu mieux, ça va donner ça.

```

<form>
  <div>
    <label>Pseudo</label>
    <input type="text" value={this.state.username} onChange={this.handlePseudo} />
  </div>

  <div>
    <label>Couleur</label>
    <select>
      <option value="vert">Vert</option>
      <option value="rouge">rouge</option>
      <option value="orange">orange</option>
    </select>
  </div>
</form>

```

On enregistre et on regarde le résultat, là on a écrit en dur mais ce qui sera bien c'est de gérer cela par le state,

```

state = {
  username: '',
  color: '',
  colorList: ["", "red", "blue", "green", "black", "pink"]
}

```

Nous allons récupérer les valeurs du tableau en utilisant la méthode map,

```
<select>
{
  this.state.colors.map(color => {
    return <option value={color}>{color}</option>
  })
}
</select>
```

Allons voir le rendu. Une fois que l'on a récupéré une data on va devoir indiquer une value,

```
<label>Couleur</label>
<select value="" onChange={this.handleClick}>
```

Nous allons devoir modifier légèrement le select car on doit indiquer un index rappeler vous on l'a déjà vu ensemble précédemment.

```
<select value="" onChange={this.handleClick}>
{
  this.state.colors.map((color, index) => {
    return <option key={index} value={color}>{color}</option>
  })
}
</select>
```

Finissons la méthode afin de gérer l'affichage,

```
handleColor = event => {
  this.setState({
    color: event.target.value
  })
}
```

Ajoutons la value,

```
<label>Couleur</label>
<select value={this.state.color} onChange={this.handleClick}>
```

Et voilà pour le select, n'hésitez pas à utiliser des console.log afin de mieux comprendre ce que l'on fait pas à pas. Passons au text-area comme tout fonctionne très bien.

Créons le text-area et ajoutons-le au state.

```
state = {
  username: '',
  color: '',
  colors: ['', 'red', 'blue', 'green', 'black', 'pink'],
  comment: ''
}
```

Nous venons de créer l'espace de commentaire, passons à la suite.

```
<div>
  <label>Commentaire</label>
  <textarea value="" onChange={this.handleComments}></textarea>
</div>
```

Maintenant que nous avons créé cela allons créer la méthode,

```
handleComments = event => {
  this.setState({
    comment: event.target.value
  })
}
```

Ajoutons-la à la value comme toujours.

```
<div>
  <label>Commentaire</label>
  <textarea value={this.state.comment} onChange={this.handleComments}>
</div>
```

Et donc pour résumé nous arrivons à mettre à jour le state au niveau du username, de la couleur et du commentaire. Comme c'est un svg ce que l'on aimera c'est changer la couleur de l'image quand on choisit une couleur dans le select. Dans le formulaire nous avons besoin de récupérer la donnée color, et ajoutons la dans le composant Car.

```
<Car color={this.state.color} height="400" />
<h1>Commentaire</h1>
```

Essayer cela fonctionne bien, on peut désormais changer la couleur de l'image grâce au select. Notre prochain objectif est de valider le formulaire et d'afficher les données au niveau de la console. Que va-t-on faire pour gérer cela ???

Un bouton et mettre le form en submit et oui les p'tits devs.

```
<form onSubmit={this.handleSubmitForm}>
```

Créons la méthode après avoir créé le bouton,

```
handleSubmitForm = e => {
  e.preventDefault();
  console.log(`Username: ${this.state.username} Couleur: ${this.state.`
```

La méthode preventDefault permet de ne pas recharger la page après validation et donc de ne pas perdre les données vues que l'on veut les afficher. Et voilà tout fonctionne parfaitement, bravo à vous. Passons au CSS que vous devez attendre avec impatience.

## CSS en React :

Dans le chapitre suivant, nous allons apprendre à faire du CSS dans une application React. Pour cela, on fera quelques exercices afin d'évoquer le Inline CSS, les CSS dans des fichiers externes (External Style Sheets), les Modules CSS, avant de terminer par Bootstrap qu'on importera dans nos dépendances.

Cependant, au moment de mon apprentissage, la dernière version Bootstrap disponible était la 4.3.1.

Donc, si vous lancez `npm install bootstrap`, vous chargerez la dernière version disponible. Actuellement (décembre 2023) on est à la version 5.3.0.

Si vous n'êtes pas très à l'aise avec Bootstrap, je vous recommande alors d'installer la même version que celle utilisée dans le cours pour que vous puissiez suivre. Pour cela, vous devez préciser la version que vous souhaitez installer. Comme ceci : `npm install bootstrap@4.3.1`

On verra également la librairie « Styled Components ». Je vous invite là aussi à installer la même version que moi : `npm install styled-components@4.3.1`

De ce fait, au moment d'aborder Bootstrap, je me focalise seulement sur l'importation du fichier **bootstrap.min.css** permettant de profiter des classes CSS.

```
import "../node_modules/bootstrap/dist/css/bootstrap.min.css";
```

Notez que vous pouvez également importer le fichier **bootstrap.min.js** pour activer les animations comme le menu hamburger, etc. Pour ce faire, vous avez deux possibilités:

1. Importer le fichier à la racine de votre application (index.js) comme ceci:

```
import "../node_modules/bootstrap/dist/js/bootstrap.min.js";
```

2. **Option 2:** Utilisez les 3 liens CDN JavaScript fournis par Bootstrap juste avant la balise de fermeture `<body>` de votre index.html.

Allez on se lance. Commençons par le inline style. Nous allons devoir ouvrir les accolades et encore des accolades pour l'objet. Prenons le h1 du Form.js et stylisons cela.

```
<h1 style={{fontSize: '50px', color: 'red'}}>Commentaire</h1>
```

Vous avez remarqué la différence avec le Html, ici on écrit en camelCase, vous pouvez aller voir le résultat. On peut l'écrire différemment afin de ne pas polluer notre code.

```
const titreRouge = {
  fontSize: '50px',
  color: 'red'
}
```

Et on rappelle notre const dans nos accolades,

```
<h1 style={titreRouge}>Commentaire</h1>
```

Ça c'est du inline Css, on peut également mettre notre Css dans un fichier à part, en.Css que nous allons ensuite importer dans notre composant pour pouvoir exploiter le style que l'on souhaite. Allons dans src et créons un fichier css je l'appellerai myCss.css, créons qlq classe simple,

```
.blue {  
  color: blue;  
}  
  
.red {  
  color: red;  
}
```

Effaçons le style inline ainsi que la const et repartons à zéro, comment déclare t-on une classe dans notre jsx ????

```
<h1 className="blue">Commentaire</h1>
```

Il se passe quoi ?? Rien c'est normal on n'a pas importé le fichier, alors importons le.

```
import './myCss.css';
```

C'est magique cela fonctionne. Imaginons que l'on veuille le faire avec des props, allons dans le composant Form dans le App.js, et faisons cela.

```
<div className="App">  
  <Form head={true}/>  
</div>
```

Créons un 'p' pour appliquer cela,

```
<p className={myClass}>je suis rouge ou bleu</p>
```

```
render() {  
  
  const myClass = this.props.head ? 'blue' : 'red';  
}
```

Vous pouvez désormais jouer avec vos connaissances car nous avons déjà beaucoup appris jusque-là pour pouvoir travailler vos styles en fonction de ce que vous souhaitez avoir en fait comme logique dans votre code. Maintenant disons que j'ai envie d'avoir deux classes, ajoutons une nouvelle classe dans notre fichier Css,

```
.bigFont {
  font-size: 40px;
  font-weight: bold;
}
```

Comment ajouter cette classe en deuxième classe ????

Et bien on va utiliser les back tic (ctrl alt et le 7)

```
<h1 className="blue">Commentaire</h1>
<p className={` ${myClass} bigFont`} bigFont>je suis rouge ou bleu</p>
```

Pourquoi le sigle dollar et bien car c'est une variable, et comme cela on peut enchaîner les classes.

On va désormais attaquer les modules en Css, créons un nouveau composant appeler MyHeaderOne.js (je sais, je ne suis pas inspiré mais il est tard lol) ou on s'occupera du h1, enlevons donc notre h1 du Form.js,

```
import React from 'react';

const MyHead = () => {
  return <h1 className="blue">Commentaire</h1>
}

export default MyHead;
```

Et comme toujours on l'importe là où il doit être, donc dans Form.js comme cela,

```
import MyHead from './myHeaderOne'
```

```
<div>
  <MyHead />
  <p className={` ${myClass} bigFont`} bigFont>
```

On peut également enlever ce qui ne nous sert plus, donc la condition red blue, ainsi que ce qui est inutile désormais sur le 'p',

```
<MyHead />
<p>je suis rouge ou bleu</p>
```

Donc là j'ai mon titre et mon paragraphe, on peut également enlever la class 'blue' au h1. Créons un autre fichier que l'on appellera myCss.module.css, dans lequel on créera une class green avec la couleur green, et évidemment on l'importe dans le fichier parent, donc Form.js.

```
import styles from './myCss.module.css';
```

Ensuite appliquons le style définit dans notre new file,

```
<div>
  <h1 className={styles.green}>Commentaire 1</h1>
  <MyHead />
</div>
```

Créons un deuxième h1, pour voir si le style s'applique bien, si on le met dans le file myHeaderOne et bien cela ne fonctionnera pas car on a importé seulement dans le parent. Très bien on a vu le inlineStyle, on a vu les stylesheet que nous avons importé c'est à dire des files Css, ensuite on a vu le cas des modules.css qui sont également intéressant, nous allons désormais aborder le cas des librairies. Voyons comment importer Bootstrap (même si je n'aime pas ça). Ou sont nos dépendances ???

Package.json exactement, enlevons ce qui ne sert plus pour la suite donc le style du h1 ainsi que le composant MyHead, donc tous les imports de style et remettons le h1 de base.

```
import React, { Component } from 'react';

class Form extends Component {

  render() {

    return(
      <div>
        <h1>Commentaire 1</h1>
        <button>Valider</button>
      </div>
    )
  }
}

export default Form;
```

On se rend sur le terminal on se met sur le bon dossier, puis on ouvre Bootstrap (getbootstrap.com) et on tape npm install [bootstrap@4.3.1](https://getbootstrap.com/docs/4.3/getting-started/introduction/) on va travailler sur cette version mais vous pouvez installer une version plus récente si besoin, là on aura tous la même, ça prend qlq minutes à installer la dépendances et on vérifie si elle est bien installée si oui on peut désormais l'utiliser dans notre appli. On doit l'importer dans le fichier index.js car c'est la racine,

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

Retournez sur l'appli il y a déjà des changements sans avoir rien fait, ils ont le pris le style par défaut de bootstrap. Allons dans le Form.js et stylisons cela, (il vous faudra connaître les classes de bootstrap dans la doc bootstrap => component vous choisissez le component que vous désirez et vous regardez la classe correspondante afin de l'appliquer et voilà). Appliquons une classe au bouton,

```
<div>
  <h1>Commentaire 1</h1>
  <button className="btn btn-danger">Valider</button>
</div>
```



Allez voir le résultat, il y aura même le hover d'appliqué. Voilà pour appliquer du bootstrap c'est aussi simple que cela, je vous laisse découvrir cela tranquillement car ce n'est pas le but de la formation mais vous savez comment procéder. Passons à styled Component.

### Librairie Styled Components : (que j'aime beaucoup)

Rendez-vous sur [www.styled-components.com](http://www.styled-components.com), l'intérêt de cette librairie, c'est qu'elle nous permet de styliser nos composants d'une manière très ciblée. (c'est là la force et l'originalité de cette librairie) Imaginez que nous ayons un composant qui contient des boutons (comme ceux de la page d'accueil par exemple) et bien au lieu d'appliquer un style disons basique comme bootstrap (qui sera chargé sur toutes les pages de notre appli qu'on l'utilise ou pas), et bien on créera du style ciblé à différents composants et on chargera ses styles si et seulement si nous avons le composant en question. Il permet de créer du CSS dynamique (style Sass ou less) comme de la programmation des variables etc... Je vous encourage vivement à visiter et lire la doc c'est vraiment très simple à utiliser, et vous en êtes largement capable. Installons la dépendance de styled components, on ouvre le terminal et on tape `npm install styled-components` et on attend, ensuite on vérifie que la dépendance est bien installée.

Enlevons la classe bootstrap et essayons cela, on met l'import comme toujours,

```
import styled from 'styled-components';
```

Et modifions ce h1,

```
const Title = styled.h1`  
  color: red;  
  font-size: 80px;  
`
```

On crée un const avec le style, si vous désirez la coloration syntaxique, on installe l'extension `vscode-styled-components` (de Julien Poissonnier) une fois installé vous aurez la coloration. Passons la const en style et hop,

```
<div>  
  <Title>Commentaire</Title>  
  <button>Valider</button>  
</div>
```

Oui h1 n'existe plus et la const devient une balise, allez voir le résultat. Il a pris le style que l'on vient de mettre avec le Styled Components, dans l'inspecteur vous verrez une classe bizarre écrite. Essayer de styliser un nouveau bouton. A vous de jouer.

Moi j'ai fait cela,

```
<div>  
  <Title>Commentaire</Title>  
  <button>Valider</button>  
  <Button>Valider 2</Button>  
</div>
```

Et pour le style, j'ai opté pour ce style là

```
const Button = styled.button`  
  
  background: black;  
  color: #ffffff;  
  padding: 12px 13px;  
  font-size: 15px;`
```

Donc nous avons appris pas mal de choses encore, le inline style, le Css en mode style sheets.css, les fichiers css, les modules ainsi que les librairies.

## Exercice 7

Vous êtes bien échauffé ??? et bien nouvel exercice eheh et peut-être un autre en suivant je vous sens bien en forme 😊.

## Exercice 8

### Correction exercice 8 :

Vous avez le fichier CustomBtn.js, à modifier ou pas, je vais partir sur du destructuring, personnellement, voici mon fichier CustomBtn.js avec des props

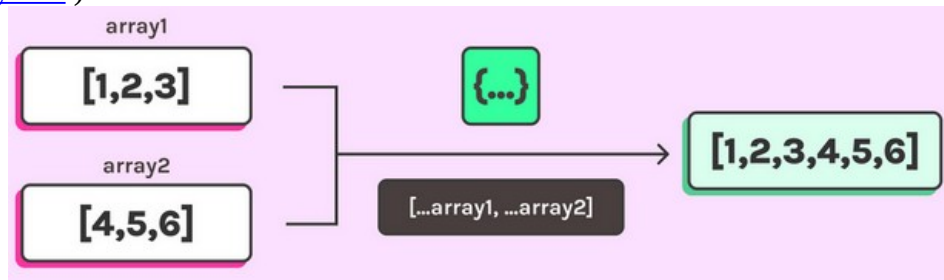
```
const Btn = ({btnStyle, children}) => {  
  
  const customBtn = {  
    backgroudColor: "grey",  
    border: 'none',  
    color: '#fff',  
    fontSize: '19px',  
    padding: '15px 30px',  
    textAlign: 'center',  
    textDecoration: 'none',  
    borderRadius: '7px',  
    display: 'block',  
    margin: '5px auto'  
  }  
}
```

Je vais utiliser le spread operator que vous ne connaissais pas à moins d'avoir cherché dans la doc, je vais donc vous montrer cela en vous expliquant ensuite.

```
return (  
  <button style={{...customBtn, ...btnStyle}}>{children}</button>  
)  
}  
  
export default Btn;
```

Def du spread operator : L'opérateur spread dans React est utilisé de la même manière qu'en JavaScript. Il fournit un moyen concis de transmettre des accessoires, de copier des objets et de gérer l'état. (exemple copie de tableau) (

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax) )



On enregistre et on importe là où on en a besoin. On peut définir le new style soit en faisant une const à l'extérieur (dans le cas d'une classe) ou dans le méthode render (qui est avant tout une fonction)

```
const success = {  
  backgroundColor: 'green',  
  color: 'black'  
}
```

Et on l'applique à notre composant

```
<Btn btnStyle={success}>Cliquez Ici</Btn>
```

On fait de même pour tous les boutons, bravo à vous si vous avez utilisé cette méthode. Vous auriez pu faire comme cela mais je n'aime pas le inline style et cela pollue notre code inutilement.

```
<Btn btnStyle={{backgroundColor: 'red', borderRadius: '20px'}}>Cliquez Ici</Btn>
```

## Exercice 9

### Correction exercice 9 :

Allons sur notre bouton donc dans le fichier CustomBtn.js, créons la méthode handleClick sans oublier le props onClick

```
return (  
  <button  
    onClick={handleClick}  
    style={{...customBtn, ...btnStyle}}  
  >{children}</button>  
)
```

Et sur App.js là où se trouve le composant Btn, on ajoute tout ça.

```
<Btn  
  handleClick={sayHello}  
  btnStyle={{backgroundColor: 'yellow', color: 'blue'}}  
>Say Hello</Btn>
```

Parfait tout fonctionne à merveille, Bravo à vous tous.

## Présentation du package React Bootstrap

Précédemment nous avons abordé le style CSS de Bootstrap appelé Vanilla Bootstrap mais pour ceux qui voudraient aller plus loin encore avec Bootstrap il existe un package React Bootstrap soit sur <https://react-bootstrap.github.io> ou bien sur <https://react-bootstrap.netlify.app/>, je ne vais pas vous expliquer comment l'utiliser ou faire une démo car le but de la formation n'est pas de travailler sur Bootstrap mais je veux que vous sachiez que cela existe, c'est une version Bootstrap qui est adaptée à React et qui vous permettra d'aller beaucoup plus loin et surtout sous forme de composant, je vous invite donc à lire la doc et de voir si cela vous intéresse (personnellement encore une fois je ne suis pas un pro Bootstrap mais vous faites vos propres choix).

## Les cycles de vie d'un composant

Attention on va aborder un sujet très très important, qui va être purement théorique, peut-être de la pratique mais pas sûr. Alors la méthode de cycle de vie d'un composant React en anglais cela s'appelle **Life Cycle Method**. Je vous invite à vous rendre sur la doc de React (reactjs.org) onglet doc et on choisit référence de l'API => react components une fois sur la page on descend légèrement au niveau du chapitre 'Le cycle de vie du composant'. On va lire ça ensemble. Regardons le diagramme. À première vue cela peut paraître compliqué mais rassurez-vous c'est très très simple, il nous indique 3 phases, mais en réalité il y en a 4, la première c'est le montage (en anglais mounting phase), la mise à jour (updating phase), et le démontage (unmounting phase). Se rajoute à ses trois phases une quatrième, ce n'est pas une phase à proprement parler on va dire, c'est deux méthodes de gestion des erreurs, mais on va pour le moment se concentrer sur ces 3 phases.

Quand on crée un composant React, on lui donne vie, c'est-à-dire que l'on va faire en sorte de l'afficher, de l'exploiter en quelques sortes. Le rendre visible sur le DOM, on va le manipuler, on va changer ses states, ses props etc.. Une fois qu'on en a plus besoin, on s'en débarrasse. Commençons par la phase de montage, on va créer notre composant sachant que les méthodes en question que nous avons ici, elles sont accessibles uniquement dans les composants de type classe. Une classe possède un constructeur pour instancier la classe en question pour en créer des objets, la méthode constructeur que nous avons ici, celle qui contient le state justement, est une méthode qui se lance systématiquement quand on instancie une classe c'est clairement indiqué dans ce diagramme, donc ça c'est la première méthode. Ensuite nous avons la méthode (ancienne version `getDerivedStateFromProps` regarder seulement la phase montage, elle est facultative) `render` au lieu de répéter cette méthode deux fois (montage et mise à jour) elle est obligatoire dans un composant de type classe, (vous avez l'habitude de l'utiliser désormais). Une fois que la méthode `render` a été appelée celle-ci va faire en sorte de mettre à jour les données sur notre dom et les refs, une fois que le dom a été mis à jour, nous avons les

datas qui s'affiche sur notre écran, on peut à ce moment-là accéder à `Component Did mount` (vous pouvez cliquer sur la checkbox en haut pour voir les anciennes méthodes) en clair ce sont les méthodes que vous allez utiliser le plus couramment. Imaginer on décide de changer le state ou le props d'un autre composant, que ce soit via la méthode `setState` (on l'a vue précédemment) ou bien on change le props ou on force la mise à jour. A ce moment-là on peut accéder(avant) à la méthode `getDerivedStateFromProps` et vérifier `shouldComponentUpdate`, est ce l'on met à jour ou pas le composant (on peut s'en passer de `shouldComponentUpdate` si on force le changement) à ce moment-là on passe directement au render, et react met à jour le dom, et c'est à ce moment-là qu'on pourra accéder à `componentDidUpdate`, ça c'est la deuxième phase ensuite viendra la phase de démontage et à ce moment-là il n'y a qu'une seule méthode celle de `componentWillUnmount`. On utilisera désormais essentiellement celle que vous voyez lorsque vous décocher la checkbox. Si on retourne sur la doc vous aurez tout ça bien détailler si vous voulez avoir un peu de lecture vous avez les 3 phases, ainsi que la gestion des erreurs. Là nous avons deux méthodes, une méthode static (ce sont des méthodes qui sont utilisées lorsque la méthode ne s'applique qu'à la classe elle-même et pas à ses instances). Savoir quand est ce que l'on utilise telle ou telle méthode est super important quand on travaille en React, et cela deviendra naturelle avec le temps. Retenez surtout les 3 phases, et n'hésitez pas à lire la doc à ce sujet.

**Fin**

