8.1. Gestion des événements

La gestion des événements avec des éléments React est très similaire à la gestion des événements sur les éléments DOM. Il existe quelques différences de syntaxe :

- Les événements React sont nommés en camelCase, plutôt qu'en minuscules.
- Avec JSX, vous transmettez une fonction en tant que callBack fonction ou arrow fonction, plutôt qu'une chaîne.

Exemples :

onClick

HTML	`<button onclick="ajouter()">`
	`  Ajouter article`
	`</button>`

REACT	`//callBack function`
	`<button onClick={ajouter}>`
	`Ajouer article`
	`</button>`

REACT	`//Arrow function`
	`<button onClick={ajouter}>`
	`Ajouer article`
	`</button>`

onSubmit

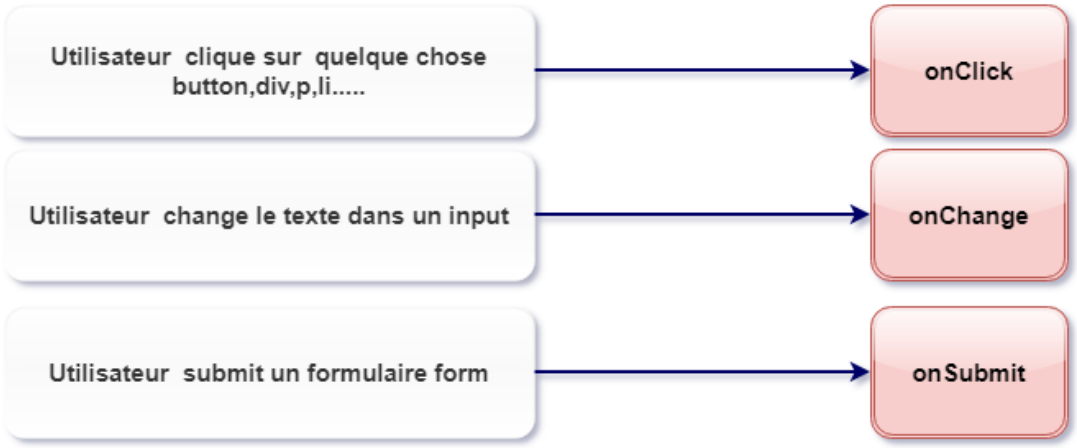
HTML	`<form onsubmit="console.log('You clicked submit.'); return false">`
	`  <button type="submit">Submit</button>`
	`</form>`

REACT	`//arrow fonction`
	`import React from 'react'`
	`export default function Form() {`
	`  function handleSubmit(e) {`
	`  e.preventDefault();console.log('You clicked submit.');`
	`  }`
	`  return (`
	`  //arrow function`
	`  <form onSubmit={(event)=>handleSubmit(event)}>`
	`  <button type="submit">Submit</button>`
	`  </form>`
	`  );`
	`  }`

REACT	//callBack function
	`import React from 'react'`
	`export default function Form() {`
	`function handleSubmit(e) {`
	`e.preventDefault();console.log('You clicked submit.');`
	`}`
	`return (`
	`//callBack function`
	`<form onSubmit={handleSubmit}>`
	`<button type="submit">Submit</button>`
	`</form>`
	`);`
	`}`

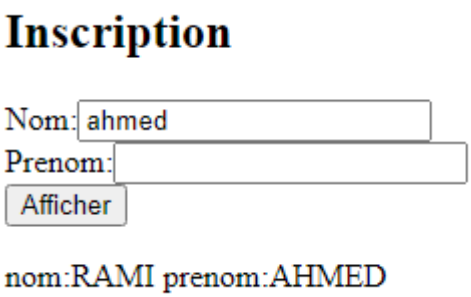
onChange

HTML	`<input type="text" onchange=”changeNom()”}></input>`
REACT	`<input type="text" onChange={changeNom}></input>`



On peut utiliser les callBack et les Arrow fonctions pour gérer les évènements

Exercice d'application 1



Quand l'utilisateur clique sur Afficher s'affiche le message contenant le nom et le prenom

Solution :

```
import React, { useState }  from "react";
export default function Inscription(){
const [nom,setNom]=useState()
const [prenom,setPrenom]=useState()
const [information,setInformation]=useState()

function envoyer(){
setInformation(`nom:${nom} prenom:${prenom}`)
}

return(
<div>
    <h2></h2>
    <div>
    <label>Nom:</label><input type="text"
onChange={(e)=>{setNom(e.target.value.toUpperCase())}}/>
    </div>
    <div>
    <label>Prenom:</label><input type="text"
onChange={(e)=>{setPrenom(e.target.value.toUpperCase())}}/>
    </div>
    <button  onClick={envoyer}>Afficher</button>
    <p>{information}</p>
</div>

)
}
```

Remarque importante : Si on remplace cette écriture

<button onClick={envoyer}>Afficher</button>

Par

<button onClick={~~envoyer()~~}>Afficher</button>

On aura le massage d'erreur

```
Uncaught Error: Too many re-renders. React limits the number of renders to prevent an infinite loop.
```

Car envoyer est exécuter après chaque render de composant, de plus la méthode envoyer contient le code suivant :

```
setInformation(`nom:${nom} prenom:${prenom}`)
```

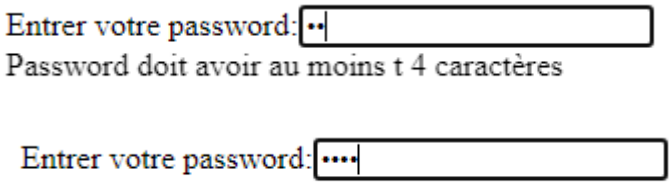
qui a son tours déclenche un render a cause de Hook **useState setInformation**

Exercice Application 2 :

On peut valider les éléments input facilement en utilisant le Système gestion d'état de Rect.

Essayons ça en créant un simple validateur de mot de passe. Ce dernier va être un input text qui nécessite l'utilisateur d'entrer un mot de passe qui contient au moins 4 caractères.

Si l'utilisateur entre un mot de passe moins de 4 caractères. un message d'erreur va être affiché «**Password doit avoir au moins de 4 caractères**"



Solution :

Avec fonctionnel composant

```
import React, { useState } from 'react'
export default function Validator(){

    const [password,setPassword]=useState('')

    return(
        <div>
            <div>
                <label>Entrer votre password:</label>
            <input type="password"
             value={password}
             onChange={(event)=>setPassword(event.target.value)}
             />
             </div>
             {password.length<4?"Password doit avoir au moins t 4
caractères":""}
        </div>
    )
}
```

Avec classe composant

```
import React, { useState } from 'react'
export default class Validator extends React.Component{

    constructor (){
        super();
        this.state={password:''}
    }
render(){

    return(
        <div>
            <div>
                <label>Entrer votre password:</label>
            <input type="password"
             value={this.state.password}
             onChange={(event)=>
this.setState({password:event.target.value})}
             />
             </div>
             {this.state.password.length<4?"Password doit avoir au moins t 4
caractères":""}
        </div>
    )
}
}
```

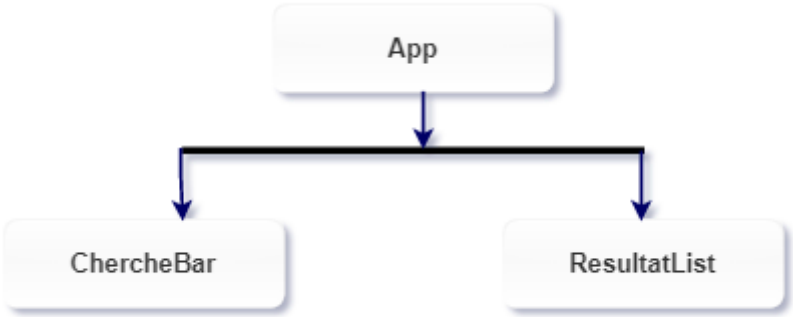
8.2. Communication inter-composant (envoi, réception de données):

Exemple :

On suppose que nous disposons de trois composants :

- App
- ChercherBar
- ResultatList

Les composants ChercherBar et ResultatList sont imbriqués dans le composant App



On dispose d'une liste

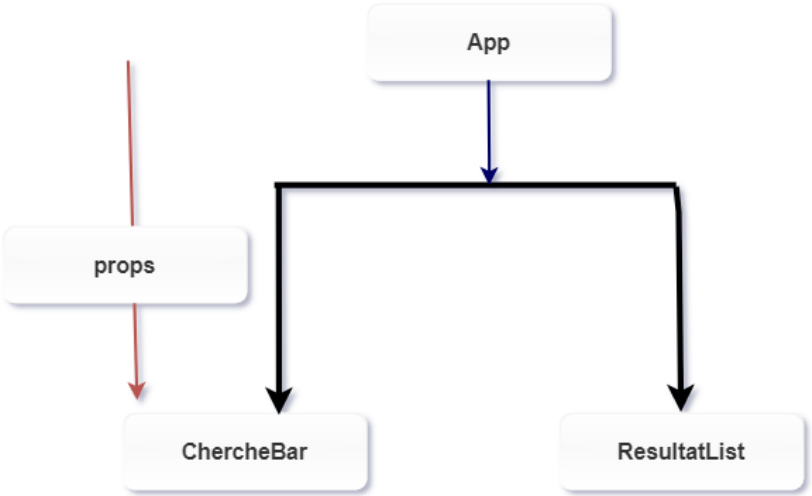
```
const list=[
{nom:"banane",type:"fruit"},
{nom:"orange",type:"fruit"},
{nom:"pomme",type:"fruit"},
{nom:"raisins",type:"fruit"},
{nom:"kiwi",type:"fruit"},
{nom:"tomate",type:"legume"},
{nom:"carotte",type:"legume"},
{nom:"pomme de terre",type:"legume"},
{nom:"navet",type:"legume"},
{nom:"poivron",type:"legume"}
]
```

Pour des raisons pédagogiques les éléments de la liste sont de type "legume" ou "fruit"

Cette liste est une constante dans le composant App

L'utilisateur saisi le type dans le composant ChercheBar, la soumission du formulaire déclenche un callBack de la fonction qui aura la valeur saisie dans Cherchebar comme argument et qui va filtrer la liste selon le type saisie, puis le composant ResultatList affiche les éléments filtrés.

8.2.1. Communication de parent vers enfant



Déjà dans la séance 6 on a vu comment passer les informations à un composant via les props

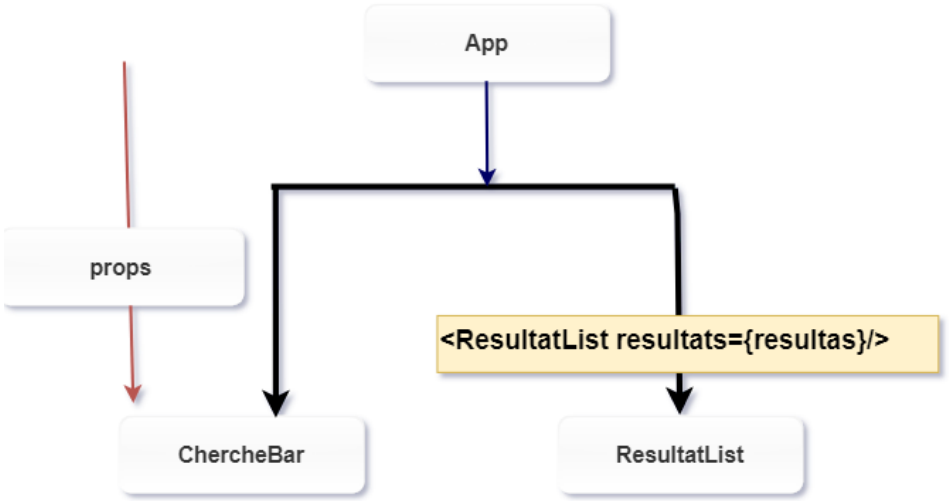
<Presentation nom="Rami" prenom="Ahmed"/>

Code App.js

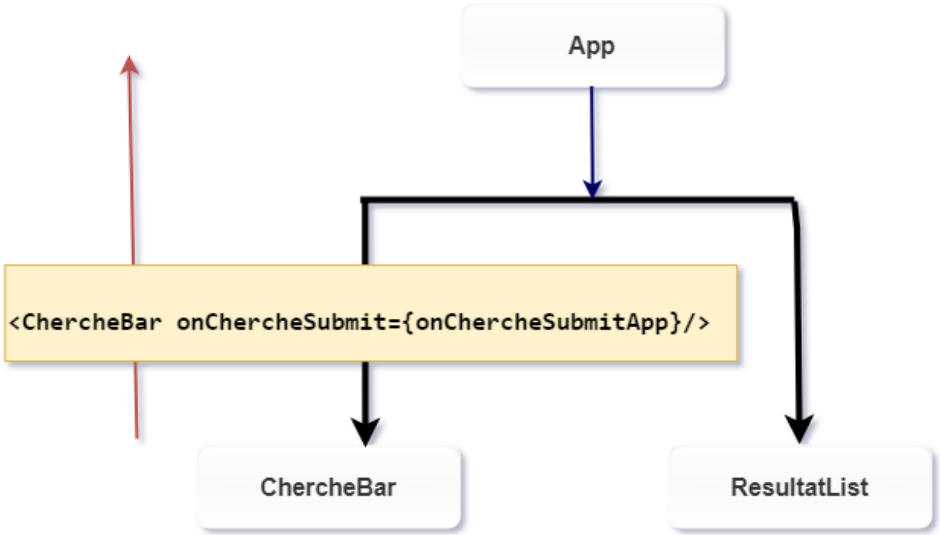
```
import React, { useState } from 'react'
import ChercheBar from './components/ChercheBar';
import ResultatList from './components/ResultatList';
const list=[
{nom:"banane",type:"fruit"},
{nom:"orange",type:"fruit"},
{nom:"pomme",type:"fruit"},
{nom:"raisins",type:"fruit"},
{nom:"kiwi",type:"fruit"},
{nom:"tomate",type:"legume"},
{nom:"carotte",type:"legume"},
{nom:"pomme de terre",type:"legume"},
{nom:"navet",type:"legume"},
{nom:"poivron",type:"legume"}
]

export default function App() {
  const [type,setType]=useState('')
  const [resultas,setResultas]=useState([])
  function onChercheSubmitApp(type){
    setType(type)
    setResultas(list.filter((item)=>item.type.toUpperCase()==type))
  }
    return (
      <div className='App'>
        <h1>Composant App</h1>
        <ChercheBar onChercheSubmit={onChercheSubmitApp}/>
        <div>
<p>le type:<span style={{color:"rgb(36,
44,33)",fontWeight:"bold"}}>{type}</span></p>
        </div>
        <ResultatList resultats={resultas}/>
      </div>
    );
  }
```

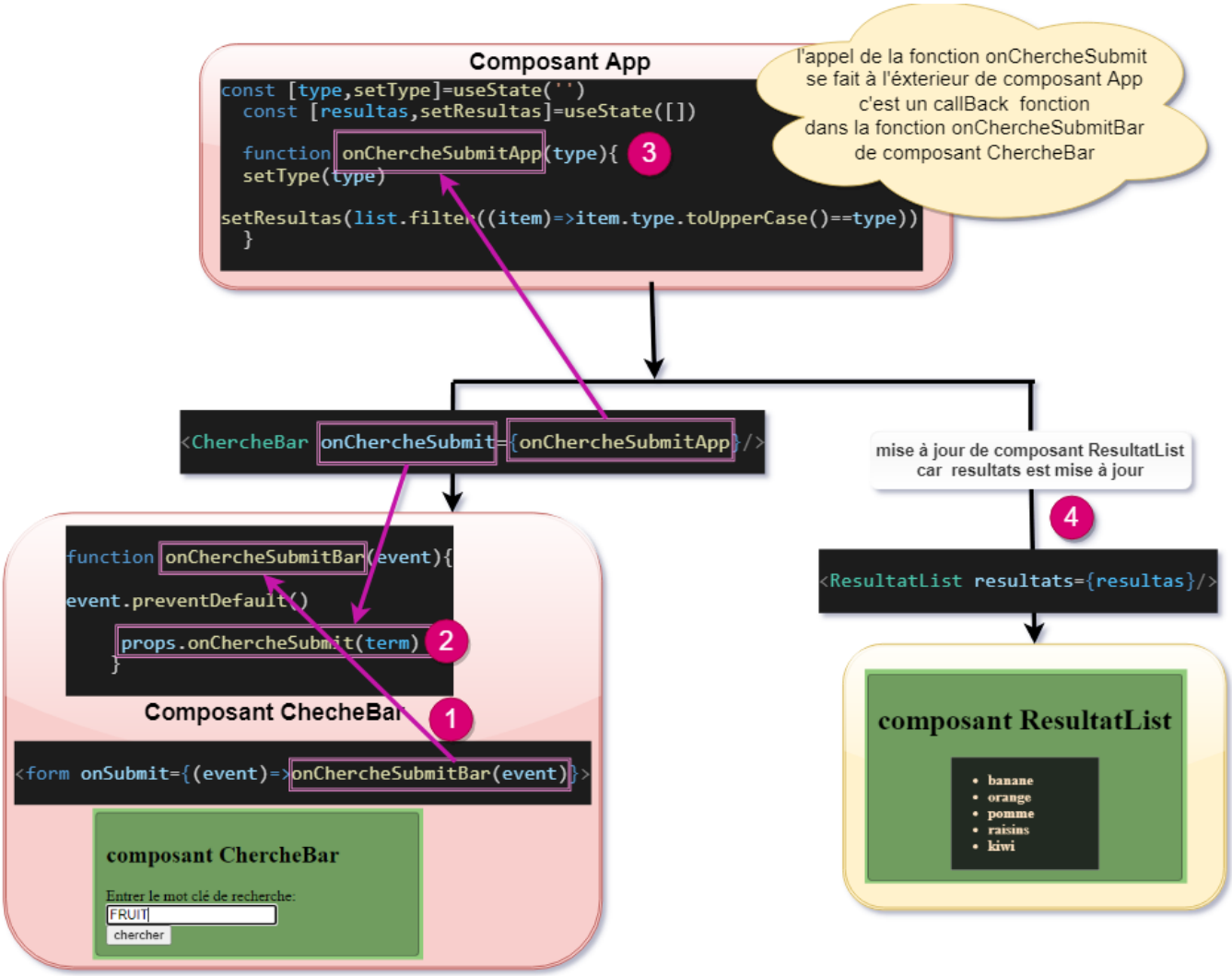
```
        <ResultatList resultats={resultas}/>
```

Le composant ResultatList reçoit l'information resultas via le props resultats



8.2.2. Communication d'enfant vers parent





Explication du code de composant App

```
<ChercheBar onChercheSubmit={onChercheSubmitApp}/>
```

On a passé au composant ChercheBar le props onChercheSubmit={onChercheSubmitApp}

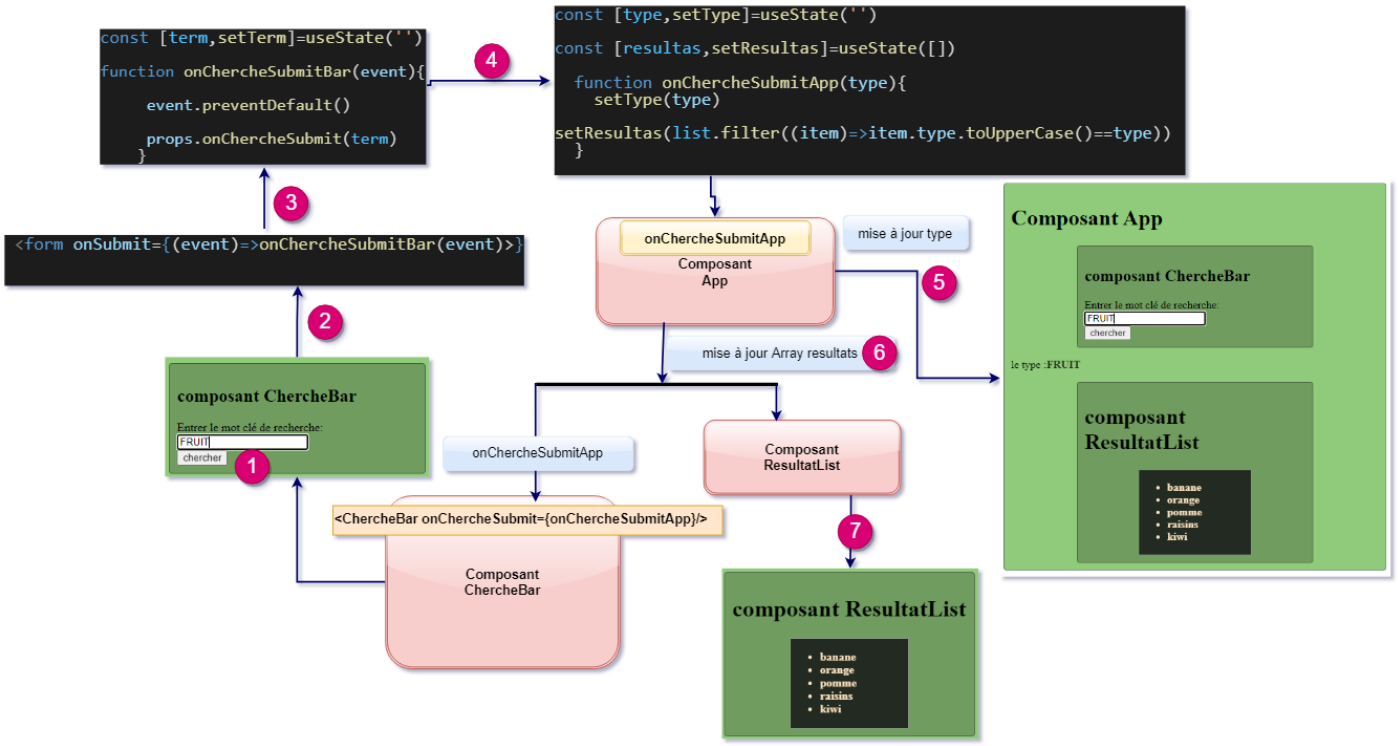
Le props onChercheSubmit contient la callback fonction onChercheSubmitApp

```
const [type,setType]=useState('')
const [resultas,setResultas]=useState([])
  function onChercheSubmitApp(type){
    setType(type)
    setResultas(list.filter((item)=>item.type.toUpperCase()==type))
  }
```

onChercheSubmitApp met à jour le type par la valeur de l'argument, puis filtre la liste sur le type dont la valeur est passée en argument.

onChercheSubmitApp sera exécuté quand l'utilisateur submit le formulaire

Le composant ResultatList est ré-rendu avec la liste résultats filtrée explication par schéma, les numéros marquent l'ordre chronologique des étapes d'exécutions



Le code de ChercheBar.js

```
import React, { useState } from "react";
export default function ChercheBar(props) {
  const [term,setTerm]=useState('')
    function onChercheSubmitBar(event){
      event.preventDefault()
     props.onChercheSubmit(term)
    }
  return (
    <div className="Child">
        <form onSubmit={(event)=>onChercheSubmitBar(event)}>
      <h2>composant ChercheBar</h2>
      <div>
        <label>Entrer le mot clé de recherche:</label>
        <input type="text" value={term}
onChange={(event)=>setTerm(event.target.value.toUpperCase())} />
      </div>
      <button type="submit">chercher</button>
      </form>
    </div>
  );
}
```

Le code de ResultatList.js

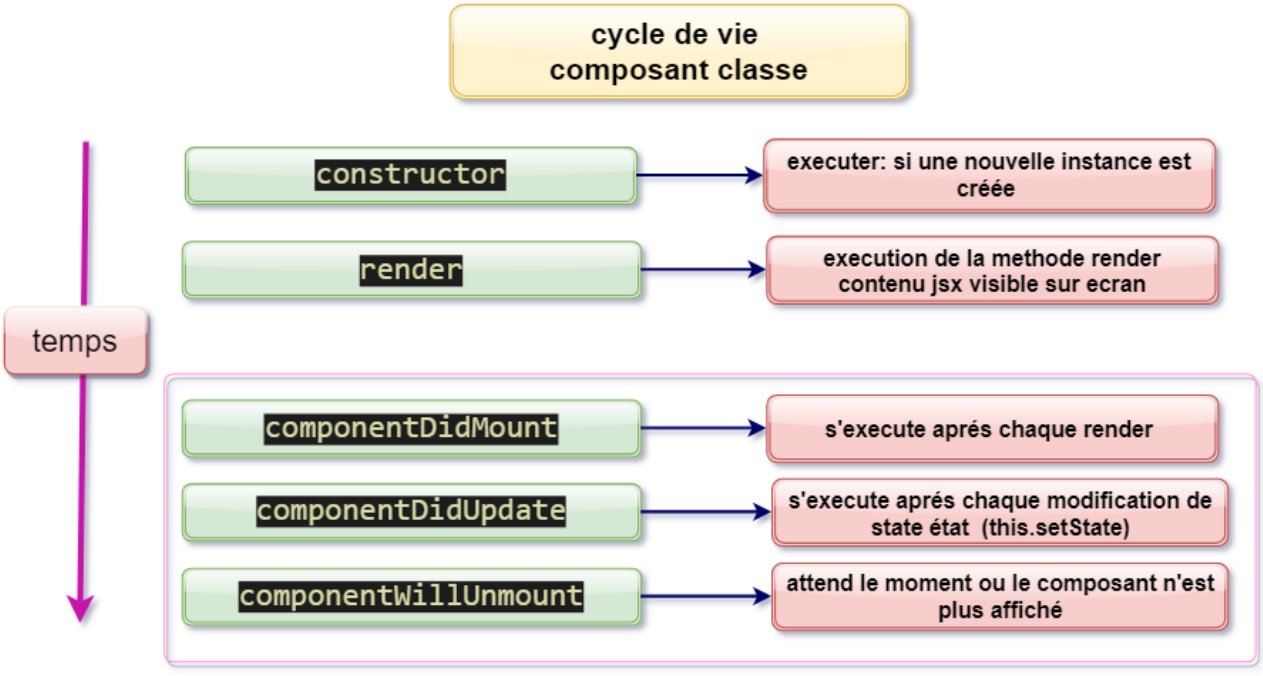
```
import React from "react";
export default function ResultatList(props) {
  return (
    <div className="Child">
      <h1>composant ResultatList</h1>
      {props.resultats.length == 0 ? (
        <p>pas de resultats</p>
      ) : (
        <div className="list">
          <ul>
            {props.resultats.map((item) => {
              return <li key={item.nom}>{item.nom}</li>;
            })}
          </ul>
        </div>
      )}
    </div>
  );
}
```

8.3. Cycle de vie des composants

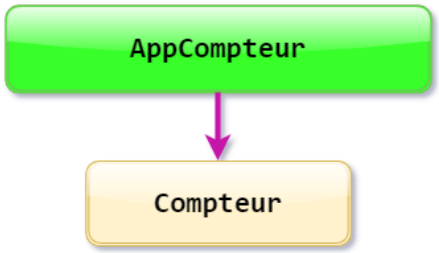
8.3.1. Cycle de vie Composants créer via classes

Composant life cycle est caractérisée par les méthodes optionnelles qu'on peut définir dans un composant créé par une classe.

React est responsable de faire l'appel automatique de ces méthodes au moment convenable (voir schéma) pendant le cycle de vie de composant



Exemple qui va illustrer le cycle de vie d'un composant classe

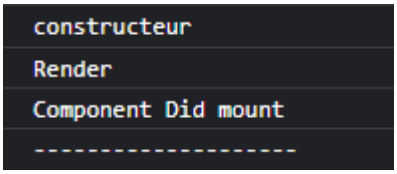


Le composant **Compteur** contient un compteur qu'on peut l'incrémenter et décrémenter

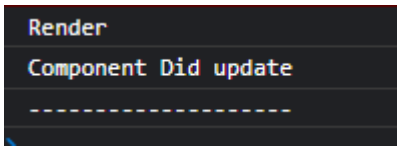
Le composant **AppCompteur** contient deux boutons pour charger et décharger le composant Compteur

Le composant **AppCompteur** est utilisé seulement pour vérifier la methode componentWillUnmount

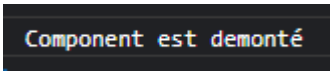
Rendu console au premier chargement



Quand je clique sur le bouton incrémenter ou décrémenter



Quand je click sur le bouton Démonter



Code source : **Index.js**

```
import React from "react";
import ReactDOM from "react-dom/client";
import AppCompteur from "./components/AppCompteur";
const root=ReactDOM.createRoot(document.getElementById("root"))
root.render(<AppCompteur/>);
```

Compteur.js

```
import React from 'react'
export default class Compteur extends React.Component{
    constructor(props) {
      console.log("constructeur")
      super(props)

      this.state = {
         compteur:0
      }
      this.incrementer=()=>{this.setState({compteur:this.state.compteur+1})}
      this.decrementer=()=>{this.setState({compteur:this.state.compteur-1})}
    }

    componentDidMount(){
      //cette methode est executé aprés render
      console.log("Component Did mount")
      console.log('--------------------')
    }

    componentDidUpdate(){
     //cette methode est executé aprés mise à jour par setState

      console.log("Component Did update")
      console.log('--------------------')
    }
componentWillUnmount(){
  console.log("Component est demonté")
}
    render(){
      console.log('Render')
      return(
        <div  style={{background:"yellow"}} >
          <h3>composant Compteur</h3>
          <p>compteur:{this.state.compteur}</p>
          <button onClick={this.incrementer}>Incrementer</button>
          <button onClick={this.decrementer}>Decrementer</button>

        </div>
      )
    }
}
```

AppCompteur.js

```
import React from 'react'
import Compteur from './Compteur'
export default class AppCompteur extends React.Component{

    constructor(props) {

      super(props)
      this.state={isMonter:true}
      this.monter=()=>{this.setState({isMonter:true})}
      this.demonter=()=>{this.setState({isMonter:false})}
    }


    render(){

      return(
        <div >
          <button onClick={this.monter}
disabled={this.state.isMonter}>Monter</button>
          <button onClick={this.demonter}
disabled={!this.state.isMonter}>Demonter</button>
         { this.state.isMonter? <Compteur/>:null}
        </div>
      )
    }
}
```

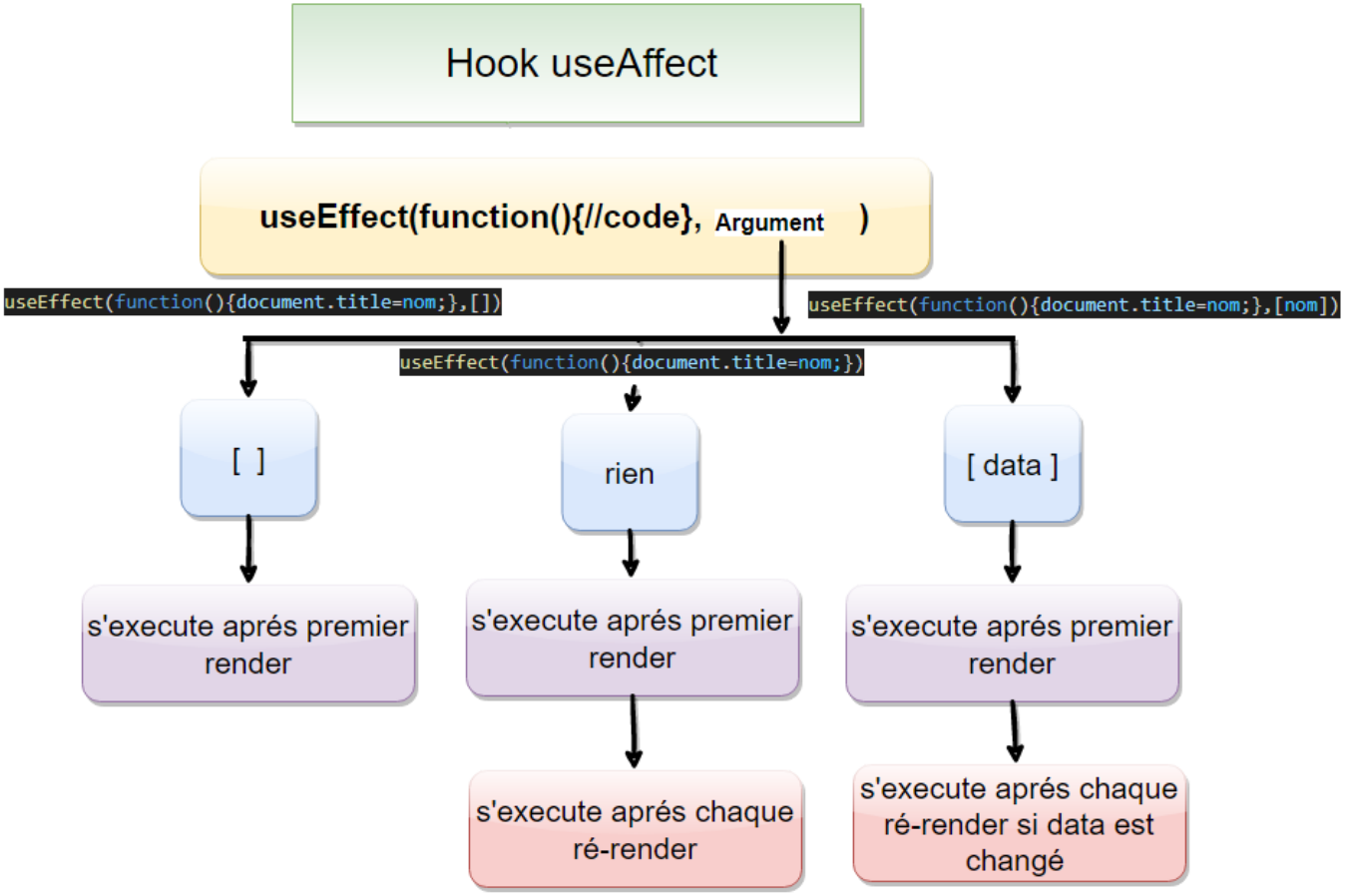
8.3.2. Cycle de vie Composants créer via fonction

Vous pouvez tirer parti du Hook **useEffect** pour obtenir les mêmes résultats qu'avec les méthodes

- componentDidMount
- componentDidUpdate
- componentWillUnmount

useEffect accepte deux paramètres. Le premier est un rappel qui s'exécute après le rendu, un peu comme dans componentDidMount.

Le deuxième paramètre est le tableau des dépendances d'effet. Si vous souhaitez l'exécuter uniquement lors du montage et du démontage, passez un tableau vide [].



Voir cours séance 12