# **Qu'est-ce que JSX ?**

JSX est une extension de syntaxe pour JavaScript. Il a été écrit pour être utilisé avec React. Le code JSX ressemble beaucoup au HTML.

# **JSX Elements**

Une unité de base de JSX est appelée un élément JSX.

Cet élément JSX ressemble exactement au HTML ! La seule différence notable est que vous le trouverez dans un fichier JavaScript, au lieu d'un fichier HTML.

# **JSX Elements And Their Surroundings**

Les éléments JSX sont traités comme des expressions JavaScript. Ils peuvent aller partout où les expressions JavaScript peuvent aller. Cela signifie qu'un élément JSX peut être enregistré dans une variable, passé à une fonction, stocké dans un objet ou un tableau... vous le nommez.

Voici un exemple d'un élément JSX enregistré dans une variable

const navBar = <nav>I am a nav bar</nav>;

Voici un exemple de plusieurs éléments JSX stockés dans un objet :

const myTeam {= {  
  center: <li>Benzo Walli</li>,  
  powerForward: <li>Rasha Loa</li>,  
  smallForward: <li>Tayshaun Dasmoto</li>,  
  shootingGuard: <li>Colmar Cumberbatch</li>,  
  pointGuard: <li>Femi Billon</li>  
};

# **Attributes In JSX**

Les éléments JSX peuvent avoir des attributs, tout comme les éléments HTML.

Un attribut JSX est écrit à l'aide d'une syntaxe de type HTML : un nom, suivi d'un signe égal, suivi d'une valeur. La valeur doit être entourée de guillemets, comme ceci :

my-attribute-name="my-attribute-value"

Voici quelques éléments JSX avec des attributs :

<a href='http://www.example.com'>Welcome to the Web</a>;  
  
const title = <h1 id='title'>Introduction to React.js: Part I</h1>;

Un seul élément JSX peut avoir de nombreux attributs, comme en HTML :

const panda = <img src='images/panda.jpg' alt='panda' width='500px' height='500px' />;

# **Nested JSX**

Vous pouvez imbriquer des éléments JSX dans d'autres éléments JSX, comme en HTML.

Voici un exemple d'élément JSX <h1>, imbriqué dans un élément JSX <a> :

Pour rendre cela plus lisible, vous pouvez utiliser des sauts de ligne et une indentation de style HTML :

<a href="https://www.example.com">  
  <h1>  
    Click me!  
  </h1>  
</a>

Si une expression JSX occupe plusieurs lignes, vous devez placer l'expression JSX multi-ligne entre parenthèses. Cela semble étrange au début, mais on s'y habitue :

(  
  <a href="https://www.example.com">  
    <h1>  
      Click me!  
    </h1>  
  </a>  
)

Les expressions JSX imbriquées peuvent être enregistrées en tant que variables, transmises à des fonctions, etc., tout comme les expressions JSX non imbriquées ! Voici un exemple d'expression JSX imbriquée enregistrée en tant que variable :

const theExample = (  
   <a href="https://www.example.com">  
     <h1>  
       Click me!  
     </h1>  
   </a>  
);

# **JSX Outer Elements**

Il existe une règle que nous n'avons pas mentionnée : une expression JSX doit avoir exactement un élément le plus externe.

En d'autres termes, ce code fonctionnera :

const paragraphs = (  
  <div id="i-am-the-outermost-element">  
    <p>I am a paragraph.</p>  
    <p>I, too, am a paragraph.</p>  
  </div>  
);

Mais ce code ne fonctionnera pas :

const paragraphs = (  
  <p>I am a paragraph.</p>   
  <p>I, too, am a paragraph.</p>  
);

La première balise d'ouverture et la dernière balise de fermeture d'une expression JSX doivent appartenir au même élément JSX !

Il est facile d'oublier cette règle et de se retrouver avec des erreurs difficiles à diagnostiquer.

Si vous remarquez qu'une expression JSX comporte plusieurs éléments externes, la solution est généralement simple : enveloppez l'expression JSX dans un <div></div>.

const blog = (

  <div>

  <img src="pics/192940u73.jpg" />

  <h1>

    Welcome to Dan's Blog!

  </h1>

  <article>

    Wow I had the tastiest sandwich today.  I <strong>literally</strong> almost freaked out.

  </article>

  </div>

);

# **Rendering JSX**

Vous avez appris à écrire des éléments JSX ! Il est maintenant temps d'apprendre à les rendre.

Restituer une expression JSX signifie la faire apparaître à l'écran.

**Question**

What’s the difference between React and ReactDOM?

**Answer**

React is a JavaScript library for building User Interfaces and ReactDOM is the JavaScript library that allows React to interact with the DOM.

# **ReactDOM.render() I**

Examinons le code que vous venez d'écrire. Commencez dans previous.js, à la ligne 5, tout à gauche.

Vous pouvez voir quelque chose appelé ReactDOM. Qu'est-ce que c'est?

ReactDOM est le nom d'une bibliothèque JavaScript. Cette bibliothèque contient plusieurs méthodes spécifiques à React, qui traitent toutes du DOM d'une manière ou d'une autre.

Nous parlerons plus tard de la façon dont ReactDOM est entré dans votre fichier. Pour l'instant, comprenez simplement que c'est à vous de l'utiliser.

Déplacez-vous légèrement vers la droite et vous pouvez voir l'une des méthodes de ReactDOM : ReactDOM.render().

ReactDOM.render() est le moyen le plus courant de rendre JSX. Il prend une expression JSX, crée un arbre correspondant de nœuds DOM et ajoute cet arbre au DOM. C'est ainsi qu'une expression JSX apparaît à l'écran.

Déplacez-vous un peu plus vers la droite, et vous arrivez à cette expression : <h1>Hello world</h1>

C'est le premier argument passé à ReactDOM.render(). Le premier argument de ReactDOM.render() doit être une expression JSX, et il sera rendu à l'écran.

Nous aborderons le deuxième argument dans le prochain exercice !

# **ReactDOM.render() II**

Déplacez-vous un peu plus vers la droite et vous verrez cette expression :

document.getElementById('app')

Vous venez d'apprendre que ReactDOM.render() fait apparaître son premier argument à l'écran. Mais où sur l'écran ce premier argument doit-il apparaître ?

Le premier argument est ajouté à tout élément sélectionné par le second argument.

Dans l'éditeur de code, sélectionnez index.html. Voyez si vous pouvez trouver un élément qui serait sélectionné par document.getElementById('app').

Cet élément a agi comme un conteneur pour le premier argument de ReactDOM.render() ! A la fin de l'exercice précédent, ceci est apparu à l'écran :

<main id="app">  
  <h1>Render me!</h1>  
</main>

# **Passing a Variable to ReactDOM.render()**

Le premier argument de ReactDOM.render() doit correspondre à une expression JSX, il ne doit pas nécessairement s'agir littéralement d'une expression JSX.

Le premier argument peut également être une variable, tant que cette variable correspond à une expression JSX.

Dans cet exemple, nous enregistrons une expression JSX en tant que variable nommée toDoList. Nous passons ensuite toDoList comme premier argument à ReactDOM.render() :

const toDoList = (  
  <ol>  
    <li>Learn React</li>  
    <li>Become a Developer</li>  
  </ol>  
);  
  
ReactDOM.render(  
  toDoList,   
  document.getElementById('app')  
);

# **The Virtual DOM**

Une particularité de ReactDOM.render() est qu'il ne met à jour que les éléments DOM qui ont changé.

Cela signifie que si vous rendez exactement la même chose deux fois de suite, le deuxième rendu ne fera rien :

const hello = <h1>Hello world</h1>;  
  
// This will add "Hello world" to the screen:  
  
ReactDOM.render(hello, document.getElementById('app'));  
  
// This won't do anything at all:  
  
ReactDOM.render(hello, document.getElementById('app'));

C'est significatif ! Seule la mise à jour des éléments DOM nécessaires est une grande partie de ce qui fait le succès de React.

React accomplit cela grâce à quelque chose appelé le DOM virtuel. Avant de passer à la fin de la leçon, lisez cet article sur le DOM virtuel.

**Article**

## Réagissez : le DOM virtuel

Lutte contre la manipulation inutile du DOM

Le problème

La manipulation DOM est au cœur du Web moderne et interactif. Malheureusement, il est également beaucoup plus lent que la plupart des opérations JavaScript.

Cette lenteur est aggravée par le fait que la plupart des frameworks JavaScript mettent à jour le DOM bien plus que nécessaire.

Par exemple, supposons que vous ayez une liste contenant dix éléments. Vous cochez le premier élément. La plupart des frameworks JavaScript reconstruiraient la liste entière. C'est dix fois plus de travail que nécessaire ! Un seul élément a changé, mais les neuf autres sont reconstruits exactement comme ils étaient avant.

Reconstruire une liste n'est pas un gros problème pour un navigateur Web, mais les sites Web modernes peuvent utiliser d'énormes quantités de manipulations DOM. La mise à jour inefficace est devenue un problème sérieux.

Pour résoudre ce problème, les gens de React ont popularisé quelque chose appelé le DOM virtuel.

Le DOM virtuel

Dans React, pour chaque objet DOM, il existe un "objet DOM virtuel" correspondant. Un objet DOM virtuel est une représentation d'un objet DOM, comme une copie allégée.

Un objet DOM virtuel a les mêmes propriétés qu'un objet DOM réel, mais il n'a pas le pouvoir de la chose réelle de changer directement ce qui est à l'écran.

La manipulation du DOM est lente. La manipulation du DOM virtuel est beaucoup plus rapide, car rien n'est dessiné à l'écran. Considérez la manipulation du DOM virtuel comme l'édition d'un plan, par opposition au déplacement de pièces dans une maison réelle.

Comment ça aide

Lorsque vous affichez un élément JSX, chaque objet DOM virtuel est mis à jour.

Cela semble incroyablement inefficace, mais le coût est insignifiant car le DOM virtuel peut se mettre à jour très rapidement.

Une fois le DOM virtuel mis à jour, React compare le DOM virtuel avec un instantané du DOM virtuel qui a été pris juste avant la mise à jour.

En comparant le nouveau DOM virtuel avec une version antérieure à la mise à jour, React détermine exactement quels objets DOM virtuels ont changé. Ce processus est appelé "différence".

Une fois que React sait quels objets DOM virtuels ont changé, React met à jour ces objets, et uniquement ces objets, sur le vrai DOM. Dans notre exemple précédent, React serait assez intelligent pour reconstruire votre seul élément de liste coché et laisser le reste de votre liste seul.

Cela fait une grande différence ! React ne peut mettre à jour que les parties nécessaires du DOM. La réputation de performance de React vient en grande partie de cette innovation.

En résumé, voici ce qui se passe lorsque vous essayez de mettre à jour le DOM dans React :

L'ensemble du DOM virtuel est mis à jour.

Le DOM virtuel est comparé à ce à quoi il ressemblait avant sa mise à jour. React détermine quels objets ont changé.

Les objets modifiés, et uniquement les objets modifiés, sont mis à jour sur le vrai DOM.

Les modifications apportées au DOM réel entraînent un changement d'écran.

Si vous souhaitez en savoir plus sur le DOM virtuel, voici un bon point de départ.

# **JSX Recap**

Toutes nos félicitations! Vous avez appris à créer et à afficher des éléments JSX ! C'est la première étape pour maîtriser React.

Dans la leçon suivante, vous apprendrez des choses puissantes que vous pouvez faire avec JSX, ainsi que certains problèmes JSX courants et comment les éviter.

**ADVANCED JSX**

# **class vs className**

Cette leçon couvrira JSX plus avancé. Vous apprendrez quelques astuces puissantes et quelques erreurs courantes à éviter.

La grammaire dans JSX est essentiellement la même que dans HTML, mais il y a des différences subtiles à surveiller. Le plus fréquent d'entre eux implique probablement le mot classe.

En HTML, il est courant d'utiliser class comme nom d'attribut : <h1 class="big">Hey</h1>

Dans JSX, vous ne pouvez pas utiliser le mot classe ! Vous devez utiliser className à la place :

<h1 className="big">Hey</h1>

En effet, JSX est traduit en JavaScript et la classe est un mot réservé en JavaScript. Lorsque JSX est rendu, les attributs JSX className sont automatiquement rendus en tant qu'attributs de classe.

## Question

Are there any other HTML attributes that are reserved keywords in JavaScript that I wont be able to use in JSX?

## Answer

In JSX, some HTML attributes will use camelCase and/or a different attribute name, but the two main attributes to be aware of (as we’ll use these most often and because their original HTML attribute name is a reserved JavaScript keyword) are:

* class - which will be className in JSX
* for - which will be htmlFor in JSX

# **Self-Closing Tags**

Un autre "gotcha" JSX implique des balises à fermeture automatique. Qu'est-ce qu'une balise à fermeture automatique ?

La plupart des éléments HTML utilisent deux balises : une balise d'ouverture (<div>) et une balise de fermeture (</div>). Cependant, certains éléments HTML tels que <img> et <input> n'utilisent qu'une seule balise. La balise qui appartient à un élément à balise unique n'est ni une balise ouvrante ni une balise fermante ; c'est une balise à fermeture automatique.

Lorsque vous écrivez une balise à fermeture automatique en HTML, il est facultatif d'inclure une barre oblique juste avant le crochet final :

Fine in HTML with a slash:

  <br />  
  
Also fine, without the slash:  
  
  <br>

Mais! Dans JSX, vous devez inclure la barre oblique. Si vous écrivez une balise à fermeture automatique dans JSX et que vous oubliez la barre oblique, vous générerez une erreur :

Fine in JSX:  
  
  <br />  
  
NOT FINE AT ALL in JSX:  
  
  <br>

Dans app.js, corrigez le JSX cassé en ajoutant des barres obliques à toutes les balises à fermeture automatique.

onst profile = (

  <div>

    <h1>I AM JENKINS</h1>

    <img src="images/jenkins.png" />

    <article>

      I LIKE TO SIT

      <br />

      JENKINS IS MY NAME

      <br />

      THANKS HA LOT

    </article>

  </div>

);

# **JavaScript In Your JSX In Your JavaScript**

Jusqu'à présent, nous nous sommes concentrés sur l'écriture d'expressions JSX. C'est similaire à l'écriture de morceaux de HTML, mais à l'intérieur d'un fichier JavaScript.

Dans cette leçon, nous allons ajouter quelque chose de nouveau : du JavaScript normal, écrit à l'intérieur d'une expression JSX, écrit à l'intérieur d'un fichier JavaScript.

## Question

Why would we want to put JavaScript in our JSX?

## Answer

We want to use JavaScript in our JSX to render logic!

When we inject JS into JSX we can make this process of rendering logic (based on things like data, events, and data changing over time) more seamless by putting our markup, the HTML part of JSX, that is based on our logic, the JS part of JSX, together in the same file.

# **Curly Braces in JSX**

Le code du dernier exercice ne s'est pas comporté comme on pourrait s'y attendre. Au lieu d'ajouter 2 et 3, il a imprimé "2 + 3" sous forme de chaîne de texte. Pourquoi? Cela s'est produit parce que 2 + 3 est situé entre les balises <h1> et </h1>.

Tout code entre les balises d'un élément JSX sera lu comme JSX, et non comme JavaScript normal ! JSX n'ajoute pas de nombres - il les lit comme du texte, tout comme HTML.

Vous avez besoin d'un moyen d'écrire du code qui dit, "Même si je suis situé entre les balises JSX, traitez-moi comme du JavaScript ordinaire et non comme du JSX." Vous pouvez le faire en enveloppant votre code entre accolades. Tout ce qui se trouve à l'intérieur des accolades sera traité comme du JavaScript normal.

## Question

Am I always going to use curly braces to write JavaScript inside of JSX?

## Answer

Yes! If we want any expression to be treated as JavaScript inside of a JSX element we need to wrap it in curly braces. Even when we assign a JavaScript expression to a variable and want to use that variable inside JSX, the variable name needs to be wrapped in curly braces.

Example:  
**app.js**:

**import** React **from** 'react';

**import** ReactDOM **from** 'react-dom';

**let** myVar = 'hello!';

**const** myJsxElement = <**h1**>myVar value is {myVar}</**h1**>; *// `myVar` without curly braces will render as `myVar` in the browser, while `{myVar}` will render as `hello!` in the browser*

ReactDOM.render(

myJsxElement,

document.getElementById('app')

);

# **20 Digits of Pi in JSX**

Vous pouvez maintenant injecter du JavaScript normal dans les expressions JSX ! Ce sera extrêmement utile. Dans l'éditeur de code, vous pouvez voir une expression JSX qui affiche les vingt premiers chiffres de pi. Étudiez l'expression et remarquez ce qui suit :

Le code est écrit dans un fichier JavaScript. Par défaut, tout sera traité comme du JavaScript normal.

Trouvez <div> sur la ligne 5. De là jusqu'à </div>, le code sera traité comme JSX.

Trouvez Mathématiques. À partir de là jusqu'à (20), le code sera à nouveau traité comme du JavaScript normal. Les accolades elles-mêmes ne seront pas traitées comme JSX ni comme JavaScript. Ce sont des marqueurs qui signalent le début et la fin d'une injection JavaScript dans JSX, similaires aux guillemets qui signalent les limites d'une chaîne.

# **Variables in JSX**

Lorsque vous injectez du JavaScript dans JSX, ce JavaScript fait partie du même environnement que le reste du JavaScript de votre fichier. Cela signifie que vous pouvez accéder aux variables à l'intérieur d'une expression JSX, même si ces variables ont été déclarées à l'extérieur.

// Declare a variable:  
const name = 'Gerdo';  
  
// Access your variable   
// from inside of a JSX expression:  
const greeting = <p>Hello, {name}!</p>;

## Question

When should I assign a variable to a JavaScript expression that I want to use in a JSX expression?

## Answer

The use of variables to store JavaScript expressions will largely be based on preference. However, we will usually want to use variables assigned to our JS expressions when our JS code would otherwise be hard to read/follow before using our JS expression inside of our JSX.  
For example:

**import** React **from** 'react';

**import** ReactDOM **from** 'react-dom';

ReactDOM.render(<**h1**>two + two = {2 + 2}</**h1**>, document.getElementById('app')); *//Here, the JavaScript expression `2+2` is easy to read and understand what's going on in the code*

vs.

**import** React **from** 'react';

**import** ReactDOM **from** 'react-dom';

**const** myFunc = (a, b) => {

*//do some logic or calculations with parameters here*

}

ReactDOM.render(<**h1**>{myFunc(3,4)}</**h1**>, document.getElementById('app')); *//Here, we assign a fu*

# **Variable Attributes in JSX**

Lors de l'écriture de JSX, il est courant d'utiliser des variables pour définir des attributs. Voici un exemple de la façon dont cela pourrait fonctionner :

// Use a variable to set the `height` and `width` attributes:  
  
const sideLength = "200px";  
  
const panda = (  
  <img   
    src="images/panda.jpg"   
    alt="panda"   
    height={sideLength}   
    width={sideLength} />  
);

Remarquez comment dans cet exemple, les attributs de <img /> ont chacun leur propre ligne. Cela peut rendre votre code plus lisible si vous avez beaucoup d'attributs sur un élément. Les propriétés d'objet sont également souvent utilisées pour définir des attributs :

const pics = {  
  panda: "http://bit.ly/1Tqltv5",  
  owl: "http://bit.ly/1XGtkM3",  
  owlCat: "http://bit.ly/1Upbczi"  
};   
  
const panda = (  
  <img   
    src={pics.panda}   
    alt="Lazy Panda" />  
);  
  
const owl = (  
  <img   
    src={pics.owl}   
    alt="Unimpressed Owl" />  
);  
  
const owlCat = (  
  <img   
    src={pics.owlCat}   
    alt="Ghastly Abomination" />  
);

import React from 'react';

import ReactDOM from 'react-dom';

const goose = 'https://content.codecademy.com/courses/React/react\_photo-goose.jpg';

// Declare new variable here:

const gooseImg = <img src={ goose }/>

ReactDOM.render(

  gooseImg,

  document.getElementById('app')

)

## Question

Is using a variable to set an attribute better practice than using object properties to set attributes?

## Answer

The decision to use a variable to set an attribute vs. using object properties to set attributes will depend on how many attributes with different values we would like to set - if we only need to set a single attribute value (or the same attribute value on multiple elements) we can easily use a variable, but if we need to set multiple attribute values, using object properties may be a better choice.

For example:

**const** myAttr = 'test'; *// we can use a variable to set a single attribute value, or multiple of the same attribute value*

**const** myElement = (

<**div**>

<**h2** className={myAttr}>A New Heading</**h2**>

<**h2** className={myAttr}>Another Heading</**h2**>

<**h2** className={myAttr}>And Another Heading</**h2**>

</**div**>

)

**const** myAttrObj = {

attrOne: 'attrOne',

attrTwo: 'attrTwo',

attrThree: 'attrThree'

}; *// we can use object properties to set multiple attribute values*

**const** myJSXElement = (

<**nav**>

<**span** id={myAttrObj.attrOne}></**span**>

<**span** id={myAttrObj.attrTwo}></**span**>

<**span** id={myAttrObj.attrThree}></**span**>

</**nav**>

)

# **Event Listeners in JSX**

Les éléments JSX peuvent avoir des écouteurs d'événement, tout comme les éléments HTML. Programmer dans React signifie travailler constamment avec des écouteurs d'événements.

Vous créez un écouteur d'événement en donnant à un élément JSX un attribut spécial. Voici un exemple:

<img onClick={myFunc} />

Le nom d'un attribut d’un event listener devrait être quelque chose comme onClick ou onMouseOver : le mot on, plus le type d'événement que vous écoutez. Vous pouvez voir une liste des noms d'événements valides ici.

La valeur d'un attribut d'écouteur d'événement doit être une fonction. L'exemple ci-dessus ne fonctionnerait que si myFunc était une fonction valide définie ailleurs :

function myFunc() {  
  alert('Make myFunc the pFunc... omg that was horrible i am so sorry');  
}  
  
<img onClick={myFunc} />

Notez qu'en HTML, les noms des Event Listener sont écrits en minuscules, comme onclick ou onmouseover. Dans JSX, les noms des écouteurs d'événements sont écrits en camelCase, comme onClick ou onMouseOver.

import React from 'react';

import ReactDOM from 'react-dom';

function makeDoggy(e) {

  // Call this extremely useful function on an <img>.

  // The <img> will become a picture of a doggy.

  e.target.setAttribute('src', 'https://content.codecademy.com/courses/React/react\_photo-puppy.jpeg');

  e.target.setAttribute('alt', 'doggy');

}

const kitty = (

  <img

    src="https://content.codecademy.com/courses/React/react\_photo-kitty.jpg"

    alt="kitty"

    onClick={makeDoggy}

  />

);

ReactDOM.render(

  kitty

, document.getElementById('app'));

# **JSX Conditionals: If Statements That Don't Work**

Bon travail! Vous avez appris à utiliser des accolades pour injecter du JavaScript dans une expression JSX ! Voici une règle que vous devez connaître : vous ne pouvez pas injecter une instruction if dans une expression JSX. Ce code cassera :

(  
  <h1>  
    {  
      if (purchase.complete) {  
        'Thank you for placing an order!'  
      }  
    }  
  </h1>  
)

La raison en est liée à la manière dont JSX est compilé. Vous n'avez pas besoin d'en comprendre les mécanismes pour l'instant, mais si vous êtes intéressé, vous pouvez en savoir plus dans la documentation de React.

Que se passe-t-il si vous souhaitez qu'une expression JSX s'affiche, mais uniquement dans certaines circonstances ? Vous ne pouvez pas injecter une instruction if. Que pouvez-vous faire?

Vous avez beaucoup d'options. Dans les prochaines leçons, nous explorerons quelques façons simples d'écrire des conditions (des expressions qui ne sont exécutées que sous certaines conditions) dans JSX.

# **JSX Conditionals: If Statements That Do Work**

Comment pouvez-vous écrire une condition, si vous ne pouvez pas injecter une instruction if dans JSX ? Eh bien, une option consiste à écrire une instruction if et à ne pas l'injecter dans JSX.

Regardez if.js. Suivez l'instruction if, de la ligne 6 à la ligne 18. if.js fonctionne, car les mots if et else ne sont pas injectés entre les balises JSX. L'instruction if est à l'extérieur et aucune injection JavaScript n'est nécessaire. C'est un moyen courant d'exprimer des conditions dans JSX.

import React from 'react';

import ReactDOM from 'react-dom';

function coinToss() {

  // This function will randomly return either 'heads' or 'tails'.

  return Math.random() < 0.5 ? 'heads' : 'tails';

}

const pics = {

  kitty: 'https://content.codecademy.com/courses/React/react\_photo-kitty.jpg',

  doggy: 'https://content.codecademy.com/courses/React/react\_photo-puppy.jpeg'

};

let img;

// if/else statement begins here:

if(coinToss() === 'heads'){

  img =

  (

    <img src={pics.kitty} />

  )

}else {

  img =

  (

     <img src={pics.doggy} />

  )

}

//ReactDOM.render()

ReactDOM.render(

  img,

  document.getElementById('app')

)

# **JSX Conditionals: The Ternary Operator**

Il existe un moyen plus compact d'écrire des conditions dans JSX : ternary operator.

Le Ternary Operator fonctionne de la même manière dans React que dans JavaScript normal. Cependant, il apparaît étonnamment souvent dans React.

Rappelez-vous comment cela fonctionne : vous écrivez x ? y : z, où x, y et z sont toutes des expressions JavaScript. Lorsque votre code est exécuté, x est évalué comme « vrai » ou « faux ». Si x est véridique, alors tout l'opérateur ternaire renvoie y. Si x est faux, alors tout l'opérateur ternaire renvoie z. Voici une belle explication si vous avez besoin d'un rappel.

Voici comment vous pouvez utiliser l'opérateur ternaire dans une expression JSX :

const headline = (  
  <h1>  
    { age >= drinkingAge ? 'Buy Drink' : 'Do Teen Stuff' }  
  </h1>  
);

Dans l'exemple ci-dessus, si l'âge est supérieur ou égal à drinkingAge, le titre sera égal à <h1>Buy Drink</h1>. Sinon, le titre sera égal à <h1>Do Teen Stuff</h1>.

'doggy'  
   
'kitty'  
   
coinToss() === 'heads'

'heads' doit renvoyer 'kitty' et 'tails' doit renvoyer 'doggy'.

import React from 'react';

import ReactDOM from 'react-dom';

function coinToss () {

  // Randomly return either 'heads' or 'tails'.

  return Math.random() < 0.5 ? 'heads' : 'tails';

}

const pics = {

  kitty: 'https://content.codecademy.com/courses/React/react\_photo-kitty.jpg',

  doggy: 'https://content.codecademy.com/courses/React/react\_photo-puppy.jpeg'

};

const img = <img src={pics[coinToss() === 'heads' ? 'kitty' : 'doggy']} />;

ReactDOM.render(

  img,

  document.getElementById('app')

);

## Question

Why can I use a ternary operator inside a JSX expression but not an if statement?

## Answer

We can use a ternary operator, also known as a conditional operator, inside a JSX expression because it will always evaluate to a value, where as an if/else/else if statement is not only not an expression (it’s a statement and will execute a statement based on the value of an expression), but will also not evaluate to a value.

In other words, we cannot use a statement where a value (or expression) is expected and for this reason, we cannot use statements, including the conditional if/else/else if statements, inside a JSX expression.

# **JSX Conditionals: &&**

Nous allons couvrir une dernière façon d'écrire des conditions dans React : l'opérateur &&. Comme l'opérateur ternaire, && n'est pas spécifique à React, mais il apparaît étonnamment souvent dans React. Comme l'opérateur ternaire, && n'est pas spécifique à React, mais il apparaît étonnamment souvent dans React.

Au cours des deux dernières leçons, vous avez écrit des déclarations qui rendaient parfois un chat et d'autres fois un chien. && n'aurait pas été le meilleur choix pour ces leçons.

&& fonctionne mieux dans les conditions qui font parfois une action, mais d'autres fois ne font rien du tout. Voici un exemple :

const tasty = (  
  <ul>  
    <li>Applesauce</li>  
    { !baby && <li>Pizza</li> }  
    { age > 15 && <li>Brussels Sprouts</li> }  
    { age > 20 && <li>Oysters</li> }  
    { age > 25 && <li>Grappa</li> }  
  </ul>  
);

Si l'expression à gauche du && est évaluée comme vraie, alors le JSX à droite du && sera rendu. Si la première expression est fausse, cependant, le JSX à droite du && sera ignoré et non rendu.

On line 13, use the && operator to make it so that this expression:

<li>Nacho Cheez Straight Out The Jar</li>

…will only appear if  !judgmental. Feel free to use the example code as a guide.

Once you click Run, then every time that you refresh the browser, there will be a 50% chance that judgmental will be true. Refresh until you see both versions of your list.

import React from 'react';

import ReactDOM from 'react-dom';

// judgmental will be true half the time.

const judgmental = Math.random() < 0.5;

const favoriteFoods = (

  <div>

    <h1>My Favorite Foods</h1>

    <ul>

      <li>Sushi Burrito</li>

      <li>Rhubarb Pie</li>

      {

       !judgmental && <li>Nacho Cheez Straight Out The

        Jar</li>

      }

      <li>Broiled Grapefruit</li>

    </ul>

  </div>

);

ReactDOM.render(

  favoriteFoods,

  document.getElementById('app')

);

# **.map in JSX**

La méthode de array .map() revient souvent dans React. Il est bon de prendre l'habitude de l'utiliser avec JSX. Si vous souhaitez créer une liste d'éléments JSX, alors .map() est souvent votre meilleur pari. Cela peut sembler odd au premier abord :

const strings = ['Home', 'Shop', 'About Me'];  
  
const listItems = strings.map(string => <li>{string}</li>);  
  
<ul>{listItems}</ul>

Dans l'exemple ci-dessus, nous commençons avec un tableau de chaînes. Nous appelons .map() sur ce tableau de chaînes, et l'appel .map() renvoie un nouveau tableau de <li>s. Sur la dernière ligne de l'exemple, notez que {listItems} sera évalué comme un tableau, car c'est la valeur renvoyée de .map() ! Les <li> JSX n'ont pas besoin d'être dans un tableau comme celui-ci, mais ils peuvent l'être.

// This is fine in JSX, not in an explicit array:  
  
<ul>  
  <li>item 1</li>  
  <li>item 2</li>  
  <li>item 3</li>  
</ul>  
  
// This is also fine!  
  
const liArray = [  
  <li>item 1</li>,   
  <li>item 2</li>,   
  <li>item 3</li>  
];  
  
<ul>{liArray}</ul>

import React from 'react';

import ReactDOM from 'react-dom';

const people = ['Rowe', 'Prevost', 'Gare'];

const peopleLis = people.map(person =>

  // expression goes here:

  <li>{person}</li>

);

// ReactDOM.render goes here:

ReactDOM.render(

  <ul>{peopleLis}</ul>,

  document.getElementById('app')

)

# **Keys**

Lorsque vous créez une liste dans JSX, votre liste devra parfois inclure quelque chose appelé keys :

<ul>  
  <li key="li-01">Example1</li>  
  <li key="li-02">Example2</li>  
  <li key="li-03">Example3</li>  
</ul>

Key est un attribut JSX. Le nom de l'attribut est key. La valeur de l'attribut doit être quelque chose d'unique, similaire à un attribut id.

Les keys ne font rien que vous puissiez voir ! React les utilise en interne pour garder une trace des listes. Si vous n'utilisez pas les clés lorsque vous êtes censé le faire, React pourrait accidentellement brouiller les éléments de votre liste dans le mauvais ordre.

Toutes les listes n'ont pas besoin d'avoir des clés. Une liste a besoin de clés si l'une des conditions suivantes est vraie : Les éléments de la liste ont de la mémoire d'un rendu à l'autre. Par exemple, lorsqu'une liste de tâches s'affiche, chaque élément doit "se souvenir" s'il a été coché. Les éléments ne doivent pas être amnésiques lorsqu'ils sont rendus.

L'ordre d'une liste peut être mélangé. Par exemple, une liste de résultats de recherche peut être mélangée d'un rendu à l'autre.

Si aucune de ces conditions n'est vraie, vous n'avez pas à vous soucier des clés. Si vous n'êtes pas sûr, cela ne fait jamais de mal de les utiliser!

import React from 'react';

import ReactDOM from 'react-dom';

const people = ['Rowe', 'Prevost', 'Gare'];

const peopleLis = people.map((person, i) =>

  // expression goes here:

  <li key={'person\_' + i}>{ person }</li>

);

// ReactDOM.render goes here:

ReactDOM.render(

  <ul>{ peopleLis }</ul>,

  document.getElementById('app')

)

# **React.createElement**

## Question

Can I use the index of an element (passed to a method/looped over) as the key attribute’s value?

## Answer

We can use the index of an element as the key attribute’s value, however, using only the index of an element as the key may cause some issues as the order of the elements may change and potentially render incorrect data specifically when a list can be reordered or filtered. When a list of elements can be reordered or filtered we should use more unique keys for the elements.

Vous pouvez écrire du code React sans utiliser JSX du tout ! La majorité des programmeurs React utilisent JSX, et nous l'utiliserons pour le reste de ce tutoriel, mais vous devez comprendre qu'il est possible d'écrire du code React sans lui.

L'expression JSX suivante : const h1 = <h1>Hello world</h1>;

peut être réécrit sans JSX, comme ceci :

const h1 = React.createElement(  
  "h1",  
  null,  
  "Hello, world"  
);

Lorsqu'un élément JSX est compilé, le compilateur transforme l'élément JSX en la méthode que vous voyez ci-dessus : React.createElement(). Chaque élément JSX est secrètement un appel à React.createElement().

Nous n'entrerons pas dans les détails du fonctionnement de React.createElement(), mais vous pouvez commencer par la documentation si vous souhaitez en savoir plus !

Take a look at the following JSX element: const greatestDivEver = <div>i am div</div>;

In **app.js**, create the element above *without* using JSX. Use the example as a guide.

const greatestDivEver = React.createElement(

  "div",

  null,

  "i am div"

);

## Question

Why would I use React.createElement() instead of JSX?

## Answer

We would use React.createElement() instead of JSX when we do not want to set up compilation for our project, which the use of JSX requires!

# **JSX Recap**

Toutes nos félicitations! Vous avez terminé l'unité sur JSX.

Vous avez appris une grande variété de concepts JSX. Si vous n'avez pas encore l'impression de les avoir tous maîtrisés, ce n'est pas grave ! Ces notions reviendront sans cesse tout au long de ce cours, et des cours suivants.

Vous êtes maintenant prêt à mettre vos connaissances JSX en pratique ! Il est temps de passer au prochain sujet majeur : React Components.

**YOUR FIRST REACT COMPONENT**

# **Hello World, Part II... THE COMPONENT**

Les applications React sont constituées de composants.

Qu'est-ce qu'un composant ?

Un composant est un petit morceau de code réutilisable qui est responsable d'un travail. Ce travail consiste souvent à rendre du HTML.

Jetez un oeil au code ci-dessous. Ce code va créer et afficher un nouveau composant React :

import React from 'react';  
import ReactDOM from 'react-dom';  
  
class MyComponentClass extends React.Component {  
  render() {  
    return <h1>Hello world</h1>;  
  }  
};  
  
ReactDOM.render(  
  <MyComponentClass />,  
  document.getElementById('app'))

Une grande partie de ce code est probablement inconnue. Cependant, vous pouvez y reconnaître du JSX, ainsi que ReactDOM.render().

Nous allons déballer ce code, un petit morceau à la fois. À la fin de cette leçon, vous comprendrez comment créer un composant React !

Question

In the context of this [lesson 135](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-react-components-u/lessons/your-first-react-component/exercises/hello-world-component), can we declare multiple components in a single file?

Answer

Yes, you can declare multiple components within a single file. For example, there can be a file containing the following components,

class Class1 extends React.Component {

render() { ... }

};

class Class2 extends React.Component {

render() { ... }

};

Choosing to have multiple components in a file or only one component per file can be up to preference. However, keeping classes in separate files can be preferred for the sake of organization.

# **Import React**

wooo! Votre premier composant de réaction ! Dans le dernier exercice, nous avons commencé par importer de react. La ligne qui a fait ça est:

import React from 'react';

Cela crée un objet nommé React qui contient les méthodes nécessaires pour utiliser la bibliothèque React.

Plus tard, nous verrons d'où la bibliothèque React est importée et comment fonctionne le processus d'importation. Pour l'instant, sachez simplement que c'est ainsi que nous importons la bibliothèque React.

Vous avez déjà vu l'une des méthodes contenues dans la bibliothèque React : React.createElement(). Rappelez-vous que lorsqu'un élément JSX est compilé, il se transforme en un appel React.createElement().

Pour cette raison, vous devez importer la bibliothèque React et l'enregistrer dans une variable nommée React avant de pouvoir utiliser n'importe quel JSX. React.createElement() doit être disponible pour que JSX fonctionne.

### Question

In the context of this [exercise 35](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-react-components-u/lessons/your-first-react-component/exercises/import-react), can you import multiple objects from a library at once?

### Answer

Yes, you can do this using “unpacking” when importing from a library.

For example, say that you wanted to import multiple functions from some library, you could do so as follows,

import { functionA, functionB, functionC } from 'library';

Another, more concrete, example is importing objects from the ‘react-router’ library.

import { Router, Switch } from 'react-router';

# **Import ReactDOM**

Afin de créer notre premier composant, nous avons ensuite importé le ReactDOM :

import ReactDOM from 'react-dom';

Cette ligne de code est très similaire à la ligne 1.

Les deux importent des objets JavaScript. Dans les deux lignes, l'objet importé contient des méthodes liées à React.

Cependant, il y a une différence!

Les méthodes importées de 'react-dom' sont destinées à interagir avec le DOM. Vous en connaissez déjà un : ReactDOM.render().

Les méthodes importées de 'react' ne traitent pas du tout du DOM, elles n'interagissent pas directement avec tout ce qui ne fait pas partie de React.

Pour clarifier : le DOM est utilisé dans les applications React, mais il ne fait pas partie de React. Après tout, le DOM est également utilisé dans d'innombrables applications non React. Les méthodes importées de 'react' sont uniquement à des fins purement React, telles que la création de composants ou l'écriture d'éléments JSX.

### Question

In the context of this [exercise 28](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-react-components-u/lessons/your-first-react-component/exercises/import-reactdom), is the ReactDOM render() method the same as the component render() method?

### Answer

No, although they have the same name, they are not the same.

The ReactDOM.render() method is used to render a React element into the DOM within the specified container, and returns a reference to that component. It is applied the following way,

ReactDOM.render(

<MyClass />,

document.getElementById('app')

);

On the other hand, the React component render() method returns the elements that make up the component. For instance,

class MyClass extends React.Component {

render() {

return <h1>Hello world</h1>;

}

}

# **Create a Component Class**

Vous avez appris qu'un composant React est un petit morceau de code réutilisable qui est responsable d'un travail, qui implique souvent le rendu HTML.

Voici un autre fait concernant les composants : nous pouvons utiliser une classe JavaScript pour définir un nouveau composant React. Nous pouvons également définir des composants avec des fonctions JavaScript, mais nous nous concentrerons d'abord sur les composants de classe.

Tous les composants de classe auront des méthodes et des propriétés en commun (nous en reparlerons plus tard). Plutôt que de réécrire ces mêmes propriétés encore et encore à chaque fois, nous étendons la classe Component de la bibliothèque React. De cette façon, nous pouvons utiliser le code que nous importons de la bibliothèque React, sans avoir à l'écrire encore et encore nous-mêmes.

Après avoir défini notre composant de classe, nous pouvons l'utiliser pour afficher autant d'instances de ce composant que nous le souhaitons.

Qu'est-ce que React.Component et comment l'utiliser pour créer une classe de composants ?

React.Component est une classe JavaScript. Pour créer votre propre classe de composant, vous devez sous-classer React.Component. Vous pouvez le faire en utilisant la classe de syntaxe YourComponentNameGoesHere étend React.Component {}.

Les classes JavaScript et les sous-classes sont un sujet complexe qui dépasse le cadre de ce cours. Si vous n'êtes pas à l'aise avec eux, voici quelques bonnes ressources à consulter : 1 2 3 4.

Jetez un autre coup d'œil au code du premier exercice :

import React from 'react';  
import ReactDOM from 'react-dom';  
  
class MyComponentClass extends React.Component {  
  render() {  
    return <h1>Hello world</h1>;  
  }  
}  
  
ReactDOM.render(  
    <MyComponentClass />,   
    document.getElementById('app')  
);

Beaucoup de choses sont encore inconnues, mais vous pouvez en comprendre plus qu'avant !

À la ligne 4, vous savez que vous déclarez une nouvelle classe de composants, qui est comme une usine pour la construction de composants React. Vous savez que React.Component est une classe que vous devez sous-classer afin de créer votre propre classe de composants. Vous savez également que React.Component est une propriété sur l'objet qui a été renvoyée par import React à partir de 'react' sur la ligne 1.

# **Name a Component Class**

Bon! Sous-classer React.Component est le moyen de déclarer une nouvelle classe de composant.

Lorsque vous déclarez une nouvelle classe de composants, vous devez donner un nom à cette classe de composants. Sur notre composant fini, le nom de notre classe de composants était MyComponentClass :

class MyComponentClass extends React.Component {  
  render() {  
    return <h1>Hello world</h1>;  
  }  
}

Les noms de variables de classe de composant doivent commencer par des lettres majuscules !

Cela respecte la syntaxe de classe de JavaScript. Il adhère également à une convention de programmation plus large dans laquelle les noms de classe sont écrits en UpperCamelCase.

De plus, il existe une raison spécifique à React pour laquelle les noms de classes de composants doivent toujours être en majuscules. Nous y reviendrons bientôt!

### Question

How should component classes be named?

### Answer

In addition to using UpperCamelCase to name component classes, there are a few other suggestions that you may consider when naming component classes.

Firstly, it can be important to name components in a meaningful and understandable way, such that the names reflect their purpose. This applies not only to React, but to any other classes and objects you name in a program.

In addition, to keep the files organized, it is good to keep only one component per file, and name files after the component they contain. So, if you named your file board.js, then the contained class is most likely named Board. This may not always be applicable, such as if a component is declared in the app.js file, however.

# **Component Class Instructions**

Passons en revue ce que vous avez appris jusqu'à présent ! Trouvez chacun de ces points dans app.js :

À la ligne 1, importer React à partir de 'react' crée un objet JavaScript. Cet objet contient les propriétés nécessaires au fonctionnement de React, telles que React.createElement() et React.Component.

À la ligne 2, importer ReactDOM à partir de 'react-dom' crée un autre objet JavaScript. Cet objet contient des méthodes qui aident React à interagir avec le DOM, telles que ReactDOM.render().

À la ligne 4, en sous-classant React.Component, vous créez une nouvelle classe de composant. Ceci n'est pas un composant ! Une classe de composants ressemble plus à une usine qui produit des composants. Lorsque vous commencerez à créer des composants, chacun proviendra d'une classe de composants.

Chaque fois que vous créez une classe de composants, vous devez donner un nom à cette classe de composants. Ce nom doit être écrit en UpperCamelCase. Dans ce cas, le nom que vous avez choisi est MyComponentClass.

Quelque chose dont nous n'avons pas encore parlé est le corps de votre classe de composant : la paire d'accolades après React.Component, et tout le code entre ces accolades.

Comme toutes les classes JavaScript, celle-ci a besoin d'un corps. Le corps agira comme un ensemble d'instructions, expliquant à votre classe de composants comment construire un composant React.

Voici à quoi ressemblerait votre corps de classe seul, sans le reste de la syntaxe de déclaration de classe. Trouvez-le dans app.js:

{  
  render() {  
    return <h1>Hello world</h1>;  
  }  
}

Cela ne ressemble pas à un ensemble d'instructions expliquant comment construire un composant React ! Pourtant c'est bien de cela qu'il s'agit.

Cliquez sur Suivant et nous verrons comment ces instructions fonctionnent.

# **The Render Function**

Une classe de composants est comme une usine qui construit des composants. Il construit ces composants en consultant un ensemble d'instructions, que vous devez fournir. Parlons de ces instructions !

Pour commencer, ces instructions doivent prendre la forme d'un corps de déclaration de classe. Cela signifie qu'ils seront délimités par des accolades, comme ceci :

class ComponentFactory extends React.Component {  
  // instructions go here, between the curly braces  
}

Les instructions doivent être écrites dans la syntaxe de classe JavaScript ES2015 typique.

Il n'y a qu'une seule propriété que vous devez inclure dans vos instructions : une méthode de rendu.

Une méthode render est une propriété dont le nom est render et dont la valeur est une fonction. Le terme "méthode de rendu" peut faire référence à l'ensemble de la propriété ou uniquement à la partie fonction.

class ComponentFactory extends React.Component {  
  render() {}  
}

Une méthode de rendu doit contenir une instruction return. Généralement, cette instruction de retour renvoie une expression JSX :

class ComponentFactory extends React.Component {  
  render() {  
    return <h1>Hello world</h1>;  
  }  
}

Bien sûr, rien de tout cela n'explique l'intérêt dqz'une méthode de rendu. Tout ce que vous savez jusqu'à présent, c'est que son nom est render, il a besoin d'une déclaration de retour pour une raison quelconque et vous devez l'inclure dans le corps de votre déclaration de classe de composant. Nous verrons bientôt le « pourquoi »!

### Question

In the context of this [exercise 10](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-react-components-u/lessons/your-first-react-component/exercises/render-function), can we return multiple elements in the render function?

### Answer

Yes, you can return more than one element in the render function.

In previous versions of React, if you wanted to do so, you had to wrap them inside another element, such as a <div> element. For example, you would have to do something like the following,

render() {

return (

<div>

<Element1 />

<Element2 />

</div>

);

}

However, the downside is that this added another element to the DOM, even if you did not want to add the <div> element.

In the current versions, you can utilize what is known as a React.Fragment, which lets you return multiple elements without the need to create a wrapper element. It would look like the following,

render() {

return (

<React.Fragment>

<Element1 />

<Element2 />

</React.Fragment>

);

}

# **Create a Component Instance**

MyComponentClass est maintenant une classe de composants de travail ! Il est prêt à suivre ses instructions et à créer des composants React. Alors, créons un composant React ! Cela ne prend qu'une ligne de plus : <MyComponentClass />\*

Pour créer un composant React, vous écrivez un élément JSX. Au lieu de nommer votre élément JSX quelque chose comme h1 ou div comme vous l'avez fait auparavant, donnez-lui le même nom qu'une classe de composant. Voilà, voilà votre instance de composant !

Les éléments JSX peuvent être soit des instances de type HTML, soit des composants. JSX utilise la capitalisation pour faire la distinction entre les deux ! C'est la raison spécifique à React pour laquelle les noms de classes de composants doivent commencer par des lettres majuscules. Dans un élément JSX, cette première lettre en majuscule indique : "Je serai une instance de composant et non une balise HTML".

### Question

In the context of this [exercise 27](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-react-components-u/lessons/your-first-react-component/exercises/create-component-instance), can we only create one component instance at a time?

### Answer

When running the ReactDOM.render() method, you can only pass in one value as the first argument, which is the component to be rendered. However, you can create more than one component instance at a time, by simply wrapping them inside of another element, such as a <div>. For example, you could do the following to render two components,

ReactDOM.render(

<div>

<ComponentOne />

<ComponentTwo />

</div>,

document.getElementById('app')

);

Another option is to create another component for the purpose of wrapping multiple components to render.

# **Render A Component**

Vous avez appris qu'une classe de composants a besoin d'un ensemble d'instructions, qui indiquent à la classe de composants comment construire des composants. Lorsque vous créez une nouvelle classe de composant, ces instructions constituent le corps de votre déclaration de classe :

class MyComponentClass extends React.Component  
{ // everything in between these curly-braces is instructions for how to build components  
  
  render() {  
    return <h1>Hello world</h1>;  
  }  
}

Cette déclaration de classe aboutit à une nouvelle classe de composant, dans ce cas nommée MyComponentClass. MyComponentClass a une méthode, nommée render. Tout cela se produit via la syntaxe de classe JavaScript standard.

Vous n'avez pas appris comment ces instructions fonctionnent réellement pour fabriquer des composants ! Lorsque vous créez un composant en utilisant l'expression <MyComponentClass />, que font ces instructions ?

Chaque fois que vous créez un composant, ce composant hérite de toutes les méthodes de sa classe de composant. MyComponentClass a une méthode : MyComponentClass.render(). Par conséquent, <MyComponentClass /> a également une méthode nommée render.

Vous pourriez créer un million d'instances <MyComponentClass /> différentes, et chacune hériterait de cette même méthode de rendu exacte.

Le dernier exercice de cette leçon consiste à rendre votre composant. Afin de rendre un composant, ce composant doit avoir une méthode nommée render. Votre composant a ceci ! Il a hérité d'une méthode nommée render de MyComponentClass.

Puisque votre composant a une méthode de rendu, tout ce qui reste à faire est de l'appeler. Cela se produit d'une manière un peu inhabituelle.

Pour appeler la méthode de rendu d'un composant, vous transmettez ce composant à ReactDOM.render(). Remarquez que votre composant est passé comme premier argument de ReactDOM.render() :

ReactDOM.render(  
  <MyComponentClass />,  
  document.getElementById('app')  
);

ReactDOM.render() indiquera à <MyComponentClass /> d'appeler sa méthode de rendu.

<MyComponentClass /> appellera sa méthode de rendu, qui renverra l'élément JSX <h1>Hello world</h1>. ReactDOM.render() prendra alors cet élément JSX résultant et l'ajoutera au DOM virtuel. Cela fera apparaître "Hello world" à l'écran.

### Question

In the context of this [lesson 18](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-react-components-u/lessons/your-first-react-component/exercises/render-react-component), what if the specified container we are rendering to already has contents?

### Answer

When the function ReactDOM.render() is first called, all contents of the container are replaced. As a result, if the container element already had some elements within it, they would be completely replaced by the new component.

For instance, say that the container element in the DOM already has contents like so,

<main id="app">

<div>Hi there!</div>

</main>

And say that we are rendering a new component into this container,

ReactDOM.render(

<MyComponentClass />,

document.getElementById('app')

);

When it renders the component, all the contents of the container will be replaced with just the <MyComponentClass /> component.

**COMPONENTS AND ADVANCED JSX**

# **Use Multiline JSX in a Component**

Dans cette leçon, vous apprendrez certaines façons courantes dont les composants JSX et React fonctionnent ensemble. Vous serez plus à l'aise avec JSX et les composants, tout en apprenant de nouvelles astuces.

Jetez un oeil à ce HTML :

<blockquote>  
  <p>  
    The world is full of objects, more or less interesting; I do not wish to add any more.  
  </p>  
  <cite>  
    <a target="\_blank"  
      href="https://en.wikipedia.org/wiki/Douglas\_Huebler">  
      Douglas Huebler  
    </a>  
  </cite>  
</blockquote>

Comment pourriez-vous créer un composant React qui restitue ce HTML ? Sélectionnez QuoteMaker.js pour voir une façon de le faire.

L'élément clé à remarquer dans QuoteMaker est les parenthèses dans l'instruction de retour, aux lignes 6 et 18. Jusqu'à présent, les instructions de retour de votre fonction de rendu ressemblaient à ceci, sans aucune parenthèse :

return <h1>Hello world</h1>;

Cependant, une expression JSX multiligne doit toujours être entourée de parenthèses ! C'est pourquoi l'instruction de retour de QuoteMaker est entourée de parenthèses.

## Question

Why not put an opening container element tag after the return keyword instead of an opening parentheses?

## Answer

Although using an opening container element tag after the return statement will work, given that the first opening tag belongs to the same element as the final closing tag of the JSX expression and the first opening tag follows the return statement on the same line, it can make our code harder to read, it can make it less organized, and if we forget to put the first opening tag on the same line as the return statement, the JavaScript engine will automatically insert a semicolon after the return statement and will not return our component, but will instead stop executing the render() function.

# **Use a Variable Attribute in a Component**

Jetez un œil à cet objet JavaScript nommé redPanda :

const redPanda = {  
  src: 'https://upload.wikimedia.org/wikipedia/commons/b/b2/Endangered\_Red\_Panda.jpg',  
  alt: 'Red Panda',  
  width: '200px  
};

Comment pourriez-vous rendre un composant React et obtenir une image avec les propriétés de redPanda ?

Sélectionnez RedPanda.js pour voir une façon de le faire.

Notez toutes les injections JavaScript d'accolades à l'intérieur de la fonction de rendu ! Les lignes 16, 17 et 18 utilisent toutes des injections JavaScript.

Vous pouvez, et le ferez souvent, injecter du JavaScript dans JSX à l'intérieur d'une fonction de rendu.

import React from 'react';

import ReactDOM from 'react-dom';

const owl = {

  title: 'Excellent Owl',

  src: 'https://content.codecademy.com/courses/React/react\_photo-owl.jpg'

};

// Component class starts here:

class Owl extends React.Component {

  render() {

    return (

      <div>

        <h1> {owl.title} </h1>

        <img

          src = {owl.src}

          alt = {owl.title} />

      </div>

    );

  }

}

ReactDOM.render(

  <Owl />,

  document.getElementById('app')

);

# **Put Logic in a Render Function**

Une fonction render() doit avoir une instruction return. Cependant, ce n'est pas tout ce qu'il peut avoir.

Une fonction render () peut également être un bon endroit pour mettre des calculs simples qui doivent se produire juste avant le rendu d'un composant. Voici un exemple de certains calculs à l'intérieur d'une fonction de rendu :

class Random extends React.Component {  
  render() {  
    // First, some logic that must happen  
    // before rendering:  
    const n = Math.floor(Math.random() \* 10 + 1);  
    // Next, a return statement  
    // using that logic:  
    return <h1>The number is {n}!</h1>;  
  }  
}

Attention à cette erreur courante :

class Random extends React.Component {  
  // This should be in the render function:  
  const n = Math.floor(Math.random() \* 10 + 1);  
  
  render() {  
    return <h1>The number is {n}!</h1>;  
  }  
};

Dans l'exemple ci-dessus, la ligne avec la déclaration const n provoquera une erreur de syntaxe, car elle ne devrait pas faire partie de la déclaration de classe elle-même, mais devrait apparaître dans une méthode comme render().

## Question

When should I use logic inside of render vs logic inside of the class declaration?

## Answer

When we have simple logic, think logic that is easy to read and understand, we can put it in our render function. If instead we have logic that we would not want to be re-created each time the render function is called, or logic that is harder to read/understand, think function definitions, event handler functions etc, we should put that logic inside the class declaration.

For example:

import React from 'react';

import ReactDOM from 'react-dom';

class MyNumbers extends React.Component {

multiplySomeNumbers(n) { //We define the `multiplySomeNumbers` method inside our class declaration because we won't want `multiplySomeNumbers` to be re-created each time the render function is called. Also, if instead we defined this logic inside of `render`, it may be hard to read/understand or convoluted.

if(n > 10) {

return n \* 5;

} else if (n > 5) {

return n \* 10;

} else {

return n \* 2

}

}

render() {

const n = Math.floor(Math.random() \* 12 + 1); //We can keep simple logic like this inside our `render` method.

return (

<div>

<h1>The starting number is {n}. After doing some calculations....the number is {this.multiplySomeNumbers(n)}. </h1>

</div>

)

}

}

# **Use a Conditional in a Render Function**

Comment pourriez-vous utiliser une instruction conditionnelle à l'intérieur d'une fonction render() ?

Sélectionnez TodaysPlan.js pour voir une façon de le faire. Notez que l'instruction if est située à l'intérieur de la fonction de rendu, mais avant l'instruction return. C'est à peu près la seule façon de voir une instruction if utilisée dans une fonction de rendu.

import React from 'react';

import ReactDOM from 'react-dom';

class TodaysPlan extends React.Component {

  render() {

    let task;

    if (!apocalypse) {

      task = 'learn React.js'

    } else {

      task = 'run around'

    }

    return <h1>Today I am going to {task}!</h1>;

  }

}

ReactDOM.render(

  <TodaysPlan />,

  document.getElementById('app')

);

## Question

Can I change a component’s attribute values using a conditional statement in the render function?

## Answer

Absolutely! We can change attribute values of components/elements based on a condition - this functionality is useful in many cases, including changing a className based on a condition.

For example, say we want to render a button and want it to have a different background color based on a condition - to add this functionality to our component we need to set the className attribute value based on said condition like this:

import React from 'react';

import ReactDOM from 'react-dom';

class MyButton extends React.Component {

render() {

let colorChoice = 'blue';

let buttonClassName;

if(colorChoice === 'red'){

buttonClassName = 'red-button';

} else {

buttonClassName = 'blue-button';

}

return (

<button className={buttonClassName}>Here's a button!</button> // the className attribute can be set by injecting JS into our component, in this example className is set based on a condition

)

}

};

ReactDOM.render(

<MyButton />,

document.getElementById('app')

);

Then in our CSS we would have something like this:

.red-button {

background-color: red;

}

.blue-button {

background-color: blue

}

# **Use this in a Component**

Le mot this est beaucoup utilisé dans React !

Vous êtes particulièrement susceptible de voir cela à l'intérieur du corps d'une déclaration de classe de composant. Voici un exemple:

class IceCreamGuy extends React.Component {  
  get food() {  
    return 'ice cream';  
  }  
  
  render() {  
    return <h1>I like {this.food}.</h1>;  
  }  
}

Dans le code, qu'est-ce que cela signifie ?

Une fois que vous avez une supposition, faites défiler vers le bas pour voir la réponse.

…

…

La réponse simple est que this fait référence à une instance de IceCreamGuy. La réponse la moins simple est que this fait référence à l'objet sur lequel cette méthode englobante, dans ce cas .render(), est appelée. Il est presque inévitable que cet objet soit une instance de IceCreamGuy, mais techniquement, il pourrait s'agir d'autre chose.

Supposons que cela se réfère à une instance de votre classe de composant, comme ce sera le cas dans tous les exemples de ce cours. IceCreamGuy a deux méthodes : .food et .render(). Puisque cela évaluera une instance de IceCreamGuy, this.food évaluera un appel de la méthode .food de IceCreamGuy. Cette méthode évaluera à son tour la chaîne "crème glacée".

Pourquoi n'avez-vous pas besoin de parenthèses après this.food ? Cela ne devrait-il pas être this.food() ?

Vous n'avez pas besoin de ces parenthèses car .food est une méthode getter. Vous pouvez le dire à partir du get dans le corps de la déclaration de classe ci-dessus.

Il n'y a rien de spécifique à React dans les méthodes getter, ni dans ce comportement ! Cependant, dans React, vous verrez cela utilisé de cette manière presque constamment.

this en JavaScript peut être un concept difficile ! Voici une bonne ressource pour comprendre cela en JavaScript.

## Question

What happens if I don’t use this when referencing a method from inside a class?

## Answer

When calling or referencing a method that’s defined on a class from inside another method on the same class, we need to use this to tell our method where to find the method we’re trying to call/reference. If we do not use this, our method will look for the method name (we’re referencing/calling) within the scope of it’s own definition and won’t be able to find it!

For example:

import React from 'react';

import ReactDOM from 'react-dom';

class NounGreeting extends React.Component {

noun() {

return 'Dog';

}

render() {

return <h1>Hello {this.noun()}.</h1>; //here `this` is refrring to the `noun` method defined on our `NounGreeting` class

}

}

ReactDOM.render(<NounGreeting />, document.getElementById('app'));

vs.:

import React from 'react';

import ReactDOM from 'react-dom';

class NounGreeting extends React.Component {

noun() {

return 'Dog';

}

render() {

return <h1>Hello {noun()}.</h1>; //here we would run into an error `'noun' is not defined` - this is because our `render` method is looking inside the scope of it's own definition for the `noun` method

}

}

ReactDOM.render(<NounGreeting />, document.getElementById('app'));

# **Use an Event Listener in a Component**

Les fonctions de rendu contiennent souvent des event Listeners. Voici un exemple d’un event Listener dans une fonction de rendu :

render() {  
  return (  
    <div onHover={myFunc}>  
    </div>  
  );  
}

Rappelez-vous qu'un handler est une fonction qui est appelée en réponse à un événement. Dans l'exemple ci-dessus, le handler est myFunc().

Dans React, vous définissez les handler en tant que méthodes sur une classe de composants. Comme ça:

class MyClass extends React.Component {  
  myFunc() {  
    alert('Stop it.  Stop hovering.');  
  }  
  
  render() {  
    return (  
      <div onHover={this.myFunc}>  
      </div>  
    );  
  }  
}

Notez que la classe du composant a deux méthodes : .myFunc() et .render(). .myFunc() est utilisé comme handler. .myFunc() sera appelée chaque fois qu'un utilisateur survole le rendu <div></div>.

## Question

Can I call multiple event handlers in response to a single event? **Answer**

By passing an anonymous function, or a named function, with multiple event handler calls as the function body, to our event listener (like onClick, onKeyUp, onChange, etc) we can call multiple event handlers in response to a single event.

For example:  
Using an anonymous function:

import React from 'react';

import ReactDOM from 'react-dom';

class MyButton extends React.Component {

eventHandler1() {

console.log('eventHandler1 called!');

}

eventHandler2() {

console.log('eventHandler2 called!');

}

render() {

return (

<button onClick={() => {

this.eventHandler1();

this.eventHandler2();

}}>Here's a button!</button> // here `onClick` is set to an anonymous function where the function body contains multiple function calls that will be triggered on the click event of the button

)

}

};

ReactDOM.render(

<MyButton />,

document.getElementById('app')

);

and using a named function:

import React from 'react';

import ReactDOM from 'react-dom';

class MyButton extends React.Component {

eventHandler1() {

console.log('eventHandler1 called!');

}

eventHandler2() {

console.log('eventHandler2 called!');

}

handleClick = () => {

this.eventHandler1();

this.eventHandler2();

}

render() {

return (

<button onClick={this.handleClick}>Here's a button!</button> // here `onClick` is set to a named function `handleClick` where the function body contains multiple function calls that will be triggered on the click event of the button

)

}

};

ReactDOM.render(

<MyButton />,

document.getElementById('app'));

# **Components Recap**

Toutes nos félicitations! Vous avez terminé l'unité sur les composants React.

Les composants React sont compliqués. Leur syntaxe est compliquée, et le raisonnement derrière leur syntaxe est particulièrement compliqué.

Vous avez beaucoup appris sur leur syntaxe et leur raisonnement. Vous avez découvert les classes de composants et les instances de composants. Vous avez découvert React.Component et les instructions que vous devez fournir à une classe de composant. Vous avez appris à importer et à rendre une instance de composant.

Vous avez été initié à certaines façons courantes d'utiliser JSX dans les composants React. Vous avez rendu des composants à l'aide d'expressions JSX multilignes, de la logique à l'intérieur de la fonction de rendu, d'une instruction conditionnelle, de this et d'un event Listener.

Vous avez passé beaucoup de temps à étudier les composants React de manière isolée ! Maintenant, il est temps de commencer à apprendre comment les composants s'intègrent dans le monde qui les entoure.

**COMPONENTS RENDER OTHER COMPONENTS**

**Components Interact**

Une application React peut contenir des dizaines, voire des centaines de composants.

Chaque composant peut être petit et relativement banal en soi. Cependant, lorsqu'ils sont combinés, ils peuvent former d'énormes écosystèmes d'informations incroyablement complexes.

En d'autres termes, les applications React sont constituées de composants, mais ce qui rend React spécial, ce ne sont pas les composants eux-mêmes. Ce qui rend React spécial, c'est la manière dont les composants interagissent.

Cette unité est une introduction **aux composants en interaction**.

# **A Component in a Render Function**

Voici une méthode .render() qui renvoie un élément JSX de type HTML :

class Example extends React.Component {  
  render() {  
    return <h1>Hello world</h1>;  
  }  
}

Vous avez vu des méthodes de rendu renvoyer des <div></div>, des <p></p> et des <h1></h1>, comme dans l'exemple ci-dessus. Les méthodes de rendu peuvent également renvoyer **un autre type de JSX : les instances de composant**.

class OMG extends React.Component {  
  render() {  
    return <h1>Whooaa!</h1>;  
  }  
}  
  
class Crazy extends React.Component {  
  render() {  
    return <OMG />;  
  }  
}

Dans l'exemple ci-dessus, la méthode de rendu de Crazy renvoie une instance de la classe de composants OMG. On pourrait dire que Crazy rend un <OMG />.

# **Apply a Component in a Render Function**

C'est un nouveau territoire ! Vous n'avez jamais vu un composant rendu par un autre composant auparavant.

Vous avez déjà vu un composant rendu auparavant, mais pas par un autre composant. Au lieu de cela, vous avez vu un composant rendu par ReactDOM.render().

Lorsqu'un composant rend un autre composant, ce qui se passe est très similaire à ce qui se passe lorsque ReactDOM.render() rend un composant.

1. Vous pouvez voir deux fichiers dans l'éditeur de code : ProfilePage.js et NavBar.js. Dans cette leçon, vous allez faire en sorte qu'une <ProfilePage /> affiche une <NavBar />.

ProfilePage.js :

1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. class ProfilePage extends React.Component {
4. render() {
5. return (
6. <div>
7. <NavBar />
8. <h1>All About Me!</h1>
9. <p>I like movies and blah blah blah blah blah</p>
10. <img src="https://content.codecademy.com/courses/React/react\_photo-monkeyselfie.jpg" />
11. </div>
12. );
13. }
14. }

NavBar.js :

import React from 'react';

class NavBar extends React.Component {

  render() {

    const pages = ['home', 'blog', 'pics', 'bio', 'art', 'shop', 'about', 'contact'];

    const navLinks = pages.map(page => {

      return (

        <a href={'/' + page}>

          {page}

        </a>

      )

    });

    return <nav>{navLinks}</nav>;

  }

}

Question

Why don’t I need a ReactDOM.render() for each component?

Answer

As we will be dealing with multiple components in React, we will be coming across with the analogy that it’s structure is like a tree you have your main trunk which is also called the root component, that is the one that will have :

ReactDOM.render(<RootComponent/>, document.getElementById('root'));

there you can see

1. we named our component RootComponent (but it is not a common practice, most likely the main component will have the name app).
2. we are rendering it in the element with id of root, this is most likely a body tag or a div tag with id of root.

Once that is set, any other component will be related to that RootComponent, so going back to the tree analogy, our trunk component will be render using ReactDOM, and so if it had three branches we would import the Branch component in it and return them in the Trunk's component’s render():

import React from 'react';

import ReactDOM from 'react-dom';

import Branch from '../Branch/Branch';

class Trunk extends React.Component {

render() {

return(

<div>

<Branch/>

<Branch/>

<Branch/>

</div>

)

}

}

ReactDOM.render(<Trunk/>, document.getElementById('root'));

There, Trunk will be rendered inside the element with id of root and an instance of branch will be rendered inside of the div from Trunk, Now what if Branch had smaller branches? They can just be returned inside of Branch, let’s say each has three too:

import React from 'react';

import ReactDOM from 'react-dom';

import SmallerBranch from '../SmallerBranch/SmallerBranch';

class Branch extends React.Component {

render() {

return(

<div>

<SmallerBranch/>

<SmallerBranch/>

<SmallerBranch/>

</div>

)

}

}

export default Branch;

Now, the last part is important, because by using the keyword export we make the content of the file we just wrote into, available for import.

The same way with smaller branches having Leaf component, and for each Leaf component, they will all be only exported so they can be rendered in other components because they all are interconnected and so we only need one component to be rendered directly on an HTML element.

**Require A File**

Lorsque vous utilisez React.js, chaque fichier JavaScript de votre application est invisible pour tous les autres fichiers JavaScript par défaut. ProfilePage.js et NavBar.js ne peuvent pas se voir.

C'est un problème!

À la ligne 9, vous venez d'ajouter une instance de la classe de composants NavBar. Mais puisque vous êtes dans ProfilePage.js, JavaScript n'a aucune idée de ce que signifie NavBar.

Si vous souhaitez utiliser une variable déclarée dans un fichier différent, tel que NavBar, vous devez importer la variable souhaitée. Pour importer une variable, vous pouvez utiliser une instruction d'importation :

import { NavBar } from './NavBar.js';

Nous avons déjà utilisé l'importation, mais pas comme ça ! Notez les différences entre la ligne de code ci-dessus et cette ligne familière : import React from 'react';

La première différence importante est les accolades autour de NavBar. Nous y reviendrons bientôt ! La deuxième différence importante concerne le contenu de la chaîne à la fin de l'instruction : 'react' vs './NavBar.js'.

Si vous utilisez une instruction import et que la chaîne à la fin commence par un point ou une barre oblique, import traitera cette chaîne comme un chemin de fichier. import suivra ce chemin de fichier et importera le fichier qu'il trouve.

Si votre chemin de fichier n'a pas d'extension de fichier, alors ".js" est supposé. Ainsi, l'exemple ci-dessus pourrait être raccourci :

import { NavBar } from './NavBar';

Question

Why sometimes import has {} and sometimes doesn’t?

Answer

The way on how we import the content from another file will always depend on how it is being exported, for example when we see:

import ComponentName from './ComponentName';

it means that whatever ComponentName is, it is a default export, which means that it is the only export of it’s kind, this approach is mainly implemented with objects, or some classes like in the case of components:

class ComponentName extend React.Component {

...

}

export default ComponentName;

Then when we see:

Import { ComponentName } from './ComponentName';

It just means that it is a named export by having the export keyword assigned to it which makes it like a property of an exported object:

export class ComponentName extends React.Component {

...

}

As a bonus, we can also import from named exports using aliases:

import \* as allComponentMethods from './ComponentName';

/\* which reads: import all under the name allComponentMethods\*/

import { ComponentName as NamedExported } from './ComponentName';

/\*here we change the name on the name export\*/

# **Export**

Bien! Vous avez appris à utiliser l'importation pour récupérer une variable à partir d'un fichier autre que le fichier en cours d'exécution.

Lorsque vous importez une variable à partir d'un fichier qui n'est pas le fichier actuel, une instruction d'importation ne suffit pas. Vous avez également besoin d'une instruction d'exportation, écrite dans l'autre fichier, exportant la variable que vous espérez saisir. L'exportation provient du système de modules d'ES6, tout comme l'importation. L'exportation et l'importation sont destinées à être utilisées ensemble, et vous voyez rarement l'une sans l'autre. Il existe plusieurs façons d'utiliser l'exportation. Dans ce cours, nous utiliserons un style appelé "exportations nommées". Voici comment fonctionnent les exportations nommées :

Dans un fichier, placez le mot-clé export juste avant quelque chose que vous souhaitez exporter. Ce quelque chose peut être n'importe quel var, let, const, fonction ou classe de niveau supérieur :

// Manifestos.js:  
  
export const faveManifestos = {  
  futurist: 'http://www.artype.de/Sammlung/pdf/russolo\_noise.pdf',  
  agile: 'https://agilemanifesto.org/iso/en/manifesto.html',  
  cyborg:   'http://faculty.georgetown.edu/irvinem/theory/Haraway-CyborgManifesto-1.pdf'  
};

Vous pouvez exporter plusieurs éléments à partir du même fichier :

// Manifestos.js:  
  
export const faveManifestos = {  
  futurist: 'http://www.artype.de/Sammlung/pdf/russolo\_noise.pdf',  
  agile:  'https://agilemanifesto.org/iso/en/manifesto.html',  
  cyborg:   'http://faculty.georgetown.edu/irvinem/theory/Haraway-CyborgManifesto-1.pdf'  
};  
  
export const alsoRan = 'TimeCube';

Dans un autre fichier, importez le nom de la variable, let, const, fonction ou classe du premier fichier :

// App.js:  
  
// Import faveManifestos and alsoRan from ./Manifestos.js:  
import { faveManifestos, alsoRan } from './Manifestos';  
  
// Use faveManifestos:  
console.log(`A Cyborg Manifesto:  ${faveManifestos.cyborg}`);

Ce style d'importation et d'exportation en JavaScript est appelé "exportations nommées". Lorsque vous utilisez des exportations nommées, vous devez toujours entourer vos noms importés d'accolades, comme :

import { faveManifestos, alsoRan } from './Manifestos';`

Question :

what would be the advantage of named exports over other exports?

Answer :

Name exports provide us the possibility to only import the necessary piece of code from another file without having to import more than needed if we were to use the default keyword, that will mean that everything in that default export will be available, on the other hand, using module.exports like it used to be before ES6+ would have a similar drawback since it will create a single object that will contain the exported content when required in a file, for example:

module.exports

const threeFunc = {};

threeFunc.funcOne = () => {...};

threeFunc.functwo = () => {...};

threeFunc.funcThree = () => {...};

module.exports = threeFunc.;

which in our importing file we will do:

const threeFunc = require('./threeFunc.js');

that will actually be like saying:

const threeFunc = {

funcOne = () => {...},

functwo = () => {...},

funcThree = () => {...}

};

the equivalent in ES6+ is export default, but now we also have easy access to a single item in another file, as we were mentioning, unsing named exports:

export const funcOne = () => {...};

export const functwo = () => {...};

export const funcThree = () => {...};

in this case, we are still exporting the three but if we only need one from the file you do not need to load all of them and delaying loading time by adding weight with the other two functions by using:

import { funcTwo } from './threeFunc';

Now, yes, we can do the same thing if we exported them all with export default and write:

import threeFunc.funcTwo from './threeFunc';

but what if we do not want to have all the functions capable to be exported:

const funcOne = () => {...};

export const functwo = () => {...};

const funcThree = () => {...};

I only want to have funcTwo to be available outside of the file, named exports provide that possibilty.

# **Component Rendering In Action**

Vous êtes maintenant prêt pour que <ProfilePage /> affiche <NavBar /> !

Il ne reste plus qu'à rendre <ProfilePage />.

Question

Where do I find the document.getElementById('app') that I am rendering onto?

Answer

In the case of [this lesson 108](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/components-render-each-other/exercises/component-render-action), you can find that clicking onto the folder icon at the top left of the middle panel will display multiple files, clicking on them opens them. If we open index.html we can see that the main tag has the id of app:

<body>

<main id="app">

</main>

<script src="https://s3.amazonaws.com/codecademy-content/courses/React/react-course-bundle.min.js"></script>

<script src="/ProfilePage.compiled.js"></script>

</body>

In our local projects using create-react-app we will notice that there is a public/ directory that contains index.html there will commonly be the <body> tag with an id of root which is what React.render() will use for our react components.

**THIS.PROPS**

# **this.props**

Auparavant, vous avez appris une manière dont les composants peuvent interagir : un composant peut rendre un autre composant. Dans cette leçon, vous apprendrez une autre manière dont les composants peuvent interagir : **un composant peut transmettre des informations à un autre composant**. Les informations transmises d'un composant à un autre sont appelées "**props**". Cliquez sur Suivant pour entrer dans props-land !

Dans cette vidéo, vous pouvez voir le composant Post transmettre un accessoire au composant Content. Le **props** contient une chaîne '../images/atom.png', qui est utilisée pour afficher une image.

### Question

In the context of this [lesson 5](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/this-props-intro), is props short for “properties”?

### Answer

Yes, props is short for the word “properties”.

One possible reason for this naming is that props are essentially JavaScript objects, and JavaScript objects contain properties. Furthermore, when we pass props from a component to a child component, the props are sent as an object containing the “properties” of the component.

# **Access a Component's props**

Chaque composant a quelque chose appelé props. Les props d'un composant sont un objet. Il contient des informations sur ce composant. Pour voir l'objet props d'un composant, vous utilisez l'expression this.props. Voici un exemple d'utilisation de this.props dans une méthode de rendu :

render() {  
  console.log("Props object comin' up!");  
  console.log(this.props);  
  console.log("That was my props object!");  
  return <h1>Hello world</h1>;  
}

La plupart des informations contenues dans this.props sont plutôt inutiles ! Mais certaines d'entre elles sont extrêmement importantes, comme vous le verrez.

Despite what you see in the browser, <PropsDisplayer />‘s props object isn’t really empty. It has some properties that JSON.stringify doesn’t detect. But even if you could see those properties, the props object still wouldn’t have much of value to show you right now.

import React from 'react';

import ReactDOM from 'react-dom';

class PropsDisplayer extends React.Component {

  render() {

    const stringProps = JSON.stringify(this.props);

 return (

      <div>

        <h1>CHECK OUT MY PROPS OBJECT</h1>

        <h2>{stringProps}</h2>

      </div>

    );

  }

}

// ReactDOM.render goes here:

ReactDOM.render(

  <PropsDisplayer myProp=”Hello” />,

  document.getElementById('app')

)

### Question

In the context of this [exercise 53](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/access-props), what does JSON.stringify do?

Answer

JSON.stringify is a method that converts a JavaScript object into a string. One use of the method is to store the object as a string in a database, and then convert back to an object when obtaining it.

For example,

JSON.stringify({ a: 1, b: 2 });

// "{"a": 1, "b": 2}"

# **Pass `props` to a Component**

Vous pouvez transmettre des informations à un composant React. Comment? En donnant à ce composant un attribut : <MyComponent foo="bar" />

Supposons que vous souhaitiez transmettre à un composant le message "Ceci est une information top secrète". Voici comment vous pourriez le faire :

<Example message="This is some top secret info." />

Comme vous pouvez le voir, pour transmettre des informations à un composant, vous avez besoin d'un nom pour les informations que vous souhaitez transmettre. Dans l'exemple ci-dessus, nous avons utilisé le nom message. Vous pouvez utiliser le nom de votre choix.

Si vous souhaitez transmettre des informations qui ne sont pas une chaîne, placez ces informations entre accolades. Voici comment passer un tableau :

<Greeting myInfo={["top", "secret", "lol"]} />

Dans cet exemple suivant, nous transmettons plusieurs informations à <Greeting />. Les valeurs qui ne sont pas des chaînes sont entourées d'accolades :

<Greeting name="Frarthur" town="Flundon" age={2} haunted={false} />

### Question

In the context of this [exercise 28](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/pass-props-component), how can we access the individual elements or values of a prop which is an array or object?

### Answer

When passing in a prop which is an array or an object, you can access the elements or values as you would normally.

For example, with a prop that is an array, you can access an element using the index as follows,

/\* A component with an array as a prop \*/

<SomeComponent arr={["A", "B"]} />

/\* To access the second element of the array \*/

this.props.arr[1]

If the prop is an object, you can access a specific value using the corresponding key,

/\* A component with an object as a prop \*/

<SomeComponent obj={{key: "value"}} />

/\* To access the value of the key \*/

this.props.obj["key"]

**THIS.PROPS**

# **Render a Component's props**

Vous venez de transmettre des informations à l'objet props d'un composant !

Vous souhaiterez souvent qu'un composant affiche les informations que vous transmettez.

Voici comment faire en sorte qu'un composant affiche des informations transmises :

1 - Trouvez la classe de composants qui va recevoir ces informations.

2 - Incluez this.props.name-of-information dans l'instruction de retour de la méthode de rendu de cette classe de composants.

import React from 'react';

import ReactDOM from 'react-dom';

class Greeting extends React.Component {

render() {

    return <h1>Hi there, {this.props.firstName}!</h1>;

  }

}

ReactDOM.render(

  <Greeting firstName='Groberta' />,

  document.getElementById('app')

);

### Question

In the context of this [exercise 13](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/render-component-props), we know that we can pass prop values inside of the curly braces, but can we also perform expressions within them?

Answer

Yes, any valid JavaScript expression can be placed inside of the curly braces. The curly braces are special characters that tell the parser that there is JavaScript code inside, and to run it as such. As a result, you can run any JavaScript expression inside of it, and it will be evaluated before the resulting HTML is returned.

For example, in the following code, the expression inside the curly braces will be evaluated as JavaScript before the entire <h1> element is returned.

return <h1>{ this.props.value \* 10 }</h1>

# **Pass props From Component To Component**

Vous avez appris à passer un props à un composant :

<Greeting firstName="Esmerelda" />

Vous avez également appris à accéder à un props passé et à l'afficher :

render() {  
  return <h1>{this.props.firstName}</h1>;  
}

L'utilisation la plus courante des props consiste à transmettre des informations à un composant, à partir d'un composant différent. Vous ne l'avez pas encore fait, mais c'est très similaire à ce que vous avez déjà vu.

Dans cet exercice, vous passerez un accessoire d'un composant à un autre.

**Une clarification curmudgeonly sur la grammaire:**

**Vous avez peut-être remarqué une utilisation lâche des mots prop et props. Malheureusement, c'est assez inévitable.**

**props est le nom de l'objet qui stocke les informations transmises. this.props fait référence à cet objet de stockage. Dans le même temps, chaque élément d'information transmis est appelé un prop. Cela signifie que les props peuvent faire référence à deux informations transmises, ou à l'objet qui stocke ces informations :(**

### Question

In the context of this [exercise 58](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/pass-props-inter-component), can props only be passed down the component tree?

### Answer

Yes, props can only be passed down the component tree, or from parent to child components.

It is not possible to pass props from a component to its parent or to its siblings within the same parent component. However, you can pass data, but not props, to the parent by linking a callback function to the child component.

# **Render Different UI Based on props**

Impressionnant! Vous avez passé un props d'un composant à un autre composant, accédé à cet props à partir du composant récepteur et l'avez rendu !

Vous pouvez faire plus avec les props que simplement les afficher. Vous pouvez également utiliser des props pour prendre des décisions.

Dans l'éditeur de code, regardez la classe du composant Welcome. Vous pouvez dire à partir de this.props.name à la ligne 5 que Welcome s'attend à recevoir une information appelée name. Cependant, Welcome ne restitue jamais cette information ! Au lieu de cela, il utilise les informations pour prendre une décision.

Les instances <Welcome /> rendront le texte WELCOME "2" MON SITE WEB BABYYY !!!!!, sauf si l'utilisateur est Mozart, auquel cas elles rendront le texte le plus respectueux, bonjour monsieur c'est vraiment super de vous rencontrer ici sur le net.

Le nom passé n'est pas affiché dans les deux cas ! Le nom est utilisé pour décider ce qui sera affiché. C'est une technique courante. Sélectionnez Home.js et regardez la classe de composant Home. Qu'est-ce que <Welcome /> affichera à l'écran ?

import React from 'react';

import ReactDOM from 'react-dom';

import { Greeting } from './Greeting';

class App extends React.Component {

  render() {

    return (

      <div>

        <h1>

          Hullo and, "Welcome to The Newzz," "On Line!"

        </h1>

        <Greeting name="Alison" signedIn={true}/>

        <article>

          Latest:  where is my phone?

        </article>

      </div>

    );

  }

}

ReactDOM.render(

  <App />,

  document.getElementById('app')

);

import React from 'react';

import ReactDOM from 'react-dom';

export class Greeting extends React.Component {

  render() {

    if (this.props.signedIn === false) {

      return <h1>GO AWAY</h1>;

    } else {

      return <h1>Hi there, {this.props.name}!</h1>;

    }

  }

}

import React from 'react';

import ReactDOM from 'react-dom';

import { Welcome } from './Welcome';

class Home extends React.Component {

  render() {

    return <Welcome name='Ludwig van Beethoven' />;

  }

}

ReactDOM.render(

  <Home />,

  document.getElementById('app')

);

import React from 'react';

export class Welcome extends React.Component {

  render() {

    if (this.props.name === 'Wolfgang Amadeus Mozart') {

      return (

        <h2>

          hello sir it is truly great to meet you here on the web

        </h2>

      );

    } else {

      return (

        <h2>

          WELCOME "2" MY WEB SITE BABYYY!!!!!

        </h2>

      );

    }

  }

}

### Question

In the context of this [exercise 79](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/render-ui-based-props), we have multiple return statements inside of render(), but is this valid?

### Answer

Yes, this is valid. Each return statement is nested inside of an if or else code block, meaning that only one of the code blocks will ever run on a render(), and so only one of the several return statements will run. The render() function has no limit to how many of these return statements can be inside of it, but only one will ever run at a time.

# **Put an Event Handler in a Component Class**

Vous pouvez, et le ferez souvent, passer des fonctions en tant qu'props. Il est particulièrement courant de transmettre des fonctions d’Event Handler.

Dans la leçon suivante, vous passerez une fonction d’Event Handler en tant que prop. Cependant, vous devez définir un Event Handler avant de pouvoir en transmettre un n'importe où. Dans cette leçon, vous allez définir une fonction d’Event Handler.

Comment définissez-vous un gestionnaire d'événements dans React ? Vous définissez un Event Handler en tant que méthode sur la classe du composant, tout comme la méthode de rendu. Jetez un oeil dans l'éditeur de code. Aux lignes 4 à 8, une méthode d’Event Handler est définie, avec une syntaxe similaire à celle de la méthode de rendu. À la ligne 12, cette méthode d’Event Handler est attachée à un événement (un événement click, dans ce cas).

import React from 'react';

class Example extends React.Component {

  handleEvent() {

    alert(`I am an event handler.

      If you see this message,

      then I have been called.`);

  }

  render() {

    return (

      <h1 onClick={this.handleEvent}>

        Hello world

      </h1>

    );

  }

}

////////////////////////////////////////////////////////////////////////////////

import React from 'react';

import ReactDOM from 'react-dom';

import { Button } from './Button';

class Talker extends React.Component {

  talk(){

    let speech = '';

    for(let i = 0; i < 10000; i++){

      speech += 'blah ';

    }

    alert(speech);

  }

  //

  render() {

    return <Button />;

  }

}

ReactDOM.render(

  <Talker />,

  document.getElementById('app')

);

///////////////////////////////////////////////////////////////////////////////

import React from 'react';

export class Button extends React.Component {

  render() {

    return (

      <button>

        Click me!

      </button>

    );

  }

}

### Question

In the context of this [exercise 22](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/event-handler-component-class), what other event handlers are there?

### Answer

React supports a large variety of event handlers for many different actions, including clipboard events such as copying and pasting, keyboard events like pressing a key, form events such as submitting a form, mouse events like the onClick event covered in this exercise, and many more.

Several common event handlers are listed as follows:

The onDoubleClick event handler is similar to onClick, but triggers when there is a double click, or two clicks in quick succession.

onDrag is another event handler used when an element or text selection is being dragged on a page.

onInput is used for when the values of input elements have been altered.

The onKeyDown event handler occurs when the user presses any keyboard key.

The onSubmit event handler triggers when a form has been submitted.

To check out all the available event handlers, you can see the official React documentation, under the SyntheticEvent section.

# **Pass an Event Handler as a prop**

Bon! Vous avez défini une nouvelle méthode sur la classe du composant Talker. Vous êtes maintenant prêt à transmettre cette fonction à un autre composant.

Vous pouvez transmettre une méthode exactement de la même manière que vous transmettez toute autre information. Voici, le puissant JavaScript.

### Question

In the context of this [exercise 18](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/event-handler-pass-prop), what happens if we pass the function with parentheses?

### Answer

Say that you passed in a function with parentheses after the function name as a prop, like the following,

render() {

return <Button press={this.press()}/>

}

This will call the press() function, and then the value passed as the prop will actually be the return value of the function rather than the function itself. By default, functions return undefined. As a result, when passing a function as a prop, there must not be any parentheses following its name.

The correct way to pass the function to the component would be,

render() {

return <Button press={this.press}/>

}

# **Receive an Event Handler as a prop**

Super! Vous venez de passer une fonction de <Talker /> à <Button />.

Dans l'éditeur de code, sélectionnez Button.js. Notez que Button affiche un élément <button></button>.

Si un utilisateur clique sur cet élément <button></button>, vous souhaitez que votre fonction de conversation transmise soit appelée.

Cela signifie que vous devez attacher talk au <button></button> en tant que Event Handler.

Comment tu fais ça? De la même manière que vous attachez un gestionnaire d'événements à un élément JSX : vous attribuez à cet élément JSX un attribut spécial. Le nom de l'attribut devrait être quelque chose comme onClick ou onHover. La valeur de l'attribut doit être le gestionnaire d'événements que vous souhaitez attacher.

1. Dans Button.js, ajoutez un attribut onClick au <button></button> de la méthode de rendu. La valeur de l'attribut onClick doit être la fonction de conversation transmise. Puisque vous avez nommé votre prop talk dans le dernier exercice, vous pouvez accéder à ce prop via this.props.talk. Cliquez sur Exécuter. Une fois le navigateur actualisé, cliquez sur le bouton. Ew, comme c'est énervant !

import React from 'react';

export class Button extends React.Component {

  render() {

    return (

      <button onClick={this.props.talk}>

        Click me!

      </button>

    );

  }

}

### Question

In the context of this [exercise 17](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/receive-event-handler-prop), can we pass multiple event handlers as props and choose a specific one to attach to a child component?

### Answer

Yes! You can pass in multiple event handlers as props to the child component, then to choose a specific one to attach to it you can utilize something like a ternary statement.

For example, say that you have a component whose button should run different code based on its state. We can pass the functions to the child component, then from the child component, use a ternary statement that will attach one of the two functions based on its state.

The parent component might look like the following, which passes two event handlers to its child component.

class Parent extends React.Component {

handleClickOne() {

// Code

}

handleClickTwo() {

// Code

}

render() {

return <Child onClickOne={this.handleClickOne}

onClickTwo={this.handleClickTwo} />;

}

}

Then, the child component will receive both event handlers, and attach one of them to itself, based on its state.

class Child extends React.Component {

render() {

return (

<button onClick={this.state.value == someValue ?

this.props.onClickOne : this.props.onClickTwo}>

Click me!

</button>

);

}

}

**handleEvent, onEvent, and this.props.onEvent**

Parlons de nommer les choses. Lorsque vous transmettez un Event Handler en tant que prop, comme vous venez de le faire, vous devez choisir deux noms.

Les deux choix de nommage se produisent dans la classe de composant parent, c'est-à-dire dans la classe de composant qui définit l’Event Handler et le transmet. Le premier nom que vous devez choisir est le nom du Event Handler lui-même.

Regardez Talker.js, lignes 6 à 12. C'est notre gestionnaire d'événements. Nous avons choisi de l'appeler parler. Le deuxième nom que vous devez choisir est le nom du prop que vous utiliserez pour passer le gestionnaire d'événements. C'est la même chose que votre nom d'attribut.

Pour notre prop name, nous avons également choisi talk, comme indiqué à la ligne 15 : return <Button talk={this.talk} />;

Ces deux noms peuvent être ce que vous voulez. Cependant, il existe une convention de dénomination qu'ils suivent souvent. Vous n'êtes pas obligé de suivre cette convention, mais vous devez la comprendre quand vous la voyez.

Voici comment fonctionne la convention de dénomination : tout d'abord, réfléchissez au type d'événement que vous écoutez. Dans notre exemple, le type d'événement était "clic".

Si vous êtes à l'écoute d'un événement « clic », vous nommez votre Event Handler handleClick. Si vous êtes à l'écoute d'un événement "keyPress", alors vous nommez votre Event Handler handleKeyPress :

class MyClass extends React.Component {  
  handleHover() {  
    alert('I am an event handler.');  
    alert('I will be called in response to "hover" events.');  
  }  
}

Votre nom de props doit être le mot on, plus votre type d'événement. Si vous écoutez un événement « clic », vous nommez votre props onClick. Si vous écoutez un événement "keyPress", alors vous nommez votre props onKeyPress :

class MyClass extends React.Component {  
  handleHover() {  
    alert('I am an event handler.');  
    alert('I will listen for a "hover" event.');  
  }  
  
  render() {  
    return <Child onHover={this.handleHover} />;  
  }  
}

**Une source majeure de confusion est le fait que des noms comme onClick ont une signification particulière, mais seulement s'ils sont utilisés sur des éléments de type HTML.**

Regardez Button.js. Lorsque vous attribuez à un <bouton></bouton> un attribut nommé onClick, le nom onClick a une signification particulière. Comme vous l'avez appris, cet attribut spécial onClick crée un écouteur d'événement, écoutant les clics sur le <bouton></bouton> :

// Button.js  
  
// The attribute name onClick  
// creates an event listner:  
<button onClick={this.props.onClick}>  
  Click me!  
</button>

Regardez maintenant Talker.js. Ici, lorsque vous donnez à <Button /> un attribut nommé onClick, alors le nom onClick ne fait rien de spécial. Le nom onClick ne crée pas d’Event Listener lorsqu'il est utilisé sur <Button /> - c'est juste un nom d'attribut arbitraire :

// Talker.js  
  
// The attribute name onClick  
// is just a normal attribute name:  
<Button onClick={this.handleClick} />

La raison en est que <Button /> n'est pas un élément JSX de type HTML ; c'est une instance de composant. Des noms comme onClick ne créent des Event Listeners que s'ils sont utilisés sur des éléments JSX de type HTML. Sinon, ce ne sont que des noms de props ordinaires.

### Question

In the context of this [exercise 12](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/handleevent-onevent-props-event), what is an HTML-like JSX element?

### Answer

An HTML-like JSX element is an element with a lower case tag name. In JSX, lower case tag names will be treated as HTML tags, while upper case tag names will be treated as a component instance.

For instance, lower case <button> is treated as an HTML element, while upper case <Button> would be treated as a component instance of the Button class.

As a consequence of this, React components must always start with a capital letter, or will otherwise be treated as an HTML element.

**this.props.children**

L'objet props de chaque composant a une propriété nommée children. this.props.children renverra tout ce qui se trouve entre les balises JSX d'ouverture et de fermeture d'un composant.

Jusqu'à présent, tous les composants que vous avez vus étaient des balises à fermeture automatique, telles que <MyComponentClass />. Ils n'ont pas à l'être ! Vous pourriez écrire <MyComponentClass></MyComponentClass>, et cela fonctionnerait toujours.

Regardez BigButton.js. Dans l'exemple 1, this.props.children de <BigButton> correspondrait au texte "Je suis un enfant de BigButton".

Dans l'exemple 2, this.props.children de <BigButton> correspondrait à un composant <LilButton />.

Dans l'exemple 3, this.props.children de <BigButton> serait égal à undefined.

Si un composant a plus d'un enfant entre ses balises JSX, alors this.props.children renverra ces enfants dans un tableau. Cependant, si un composant n'a qu'un seul enfant, alors this.props.children renverra l'enfant unique, non enveloppé dans un tableau.

1.Sélectionnez App.js. Notez que App rend deux instances <List></List> et que chaque <List></List> a au moins un enfant <li></li>.

Ouvrez maintenant List.js et examinez la classe de composants List. Pensez au fait que chaque instance de List va être rendue avec deux balises JSX :

<List>  // opening tag  
</List> // closing tag

…et qu'il y aura au moins un enfant <li></li> entre ces balises :

<List>  // opening tag  
  <li></li> // child  
</List> // closing tag

2.Vous pouvez voir deux titres de liste dans le navigateur, mais aucun élément de liste ! Comment pouvez-vous faire apparaître les éléments de la liste ? Dans List.js, dans la fonction de rendu, entre les balises <ul></ul>, ajoutez {this.props.children}.

import React from 'react';

export class List extends React.Component {

  render() {

    let titleText = `Favorite ${this.props.type}`;

    if (this.props.children instanceof Array) {

      titleText += 's';

    }

    return (

      <div>

        <h1>{titleText}</h1>

        <ul>{this.props.children}</ul>

      </div>

    );

  }

}

import React from 'react';

import ReactDOM from 'react-dom';

import { List } from './List';

class App extends React.Component {

  render() {

    return (

      <div>

        <List type='Living Musician'>

          <li></li>

        </List>

        <List type='Living Cat Musician'>

          <li></li>

        </List>

        <List type='Living Cat Piano'>

          <li></li>

        </List>

      </div>

    );

  }

}

ReactDOM.render(

  <App />,

  document.getElementById('app')

);

### Question

In this [exercise 20](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/this-props-children), we learned how to obtain all the children of a component at once using this.props.children. However, is there a way to obtain or work with the individual children of a component?

### Answer

Yes, React provides a helpful API for working with a component’s children through React.Children. Some of the helpful functionality it provides are as follows:

You can turn the data structure returned by this.props.children into a flat array with keys assigned to each child, using React.Children.toArray(). This is one of the common methods to be able to access each individual child of the component.

You can also map a function on each individual child using React.Children.forEach().

And, you can also get the number of children of the component using React.Children.count().

In addition to these, there are a few other useful methods provided by React.Children, which you can check out in the official documentation.

# **defaultProps**

Jetez un œil à la classe de composants Button. Notez qu'à la ligne 8, Button s'attend à recevoir un props nommé text. Le texte reçu sera affiché à l'intérieur d'un élément <button></button>.

Que se passe-t-il si personne ne transmet de texte à Button ?

Si personne ne passe de texte à Button, l'affichage de Button sera vide. Ce serait mieux si Button pouvait afficher un message par défaut à la place.

Vous pouvez y arriver en donnant à votre classe de composant une propriété nommée defaultProps :

class Example extends React.Component {  
  render() {  
    return <h1>{this.props.text}</h1>;  
  }  
}  
  
Example.defaultProps;

La propriété defaultProps doit être égale à un objet :

class Example extends React.Component {  
  render() {  
    return <h1>{this.props.text}</h1>;  
  }  
}  
  
// Set defaultProps equal to an object:  
Example.defaultProps = {};

À l'intérieur de cet objet, écrivez des propriétés pour tous les props par défaut que vous souhaitez définir :

class Example extends React.Component {  
  render() {  
    return <h1>{this.props.text}</h1>;  
  }  
}  
  
Example.defaultProps = { text: 'yo' };

Si un <Example /> ne reçoit aucun texte, il affichera "yo". Si un <Exemple /> reçoit du texte, il affichera ce texte transmis.

### Question

In the context of this [exercise 27](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-props/exercises/defaultprops), should we always set defaultProps for every prop value of the component?

### Answer

You do not always have to set defaultProps for every single prop value of a component.

In some cases, it can be important to do so. defaultProps should be used for any props that are required for the component, such as an input value. Also, props that will cause errors if they are missing a value, or cause the display to appear unsatisfactorily should be given a default value so that the page can still run smoothly.

# **this.props Recap**

Cela termine notre leçon sur les props. Excellent travail de s'y tenir !

Voici quelques-unes des compétences que vous avez acquises :

* Passer un props en donnant un attribut à une instance de composant
* Accéder à un prop transmis via this.props.prop-name
* Affichage d'un props.
* Utiliser un props pour prendre des décisions sur ce qu'il faut afficher
* Définition d'un Event Handler dans une classe de composant
* Passer un Event Handler comme props
* Recevoir un Event Handler prop et l'attacher à un Event Listener
* Nommer les Event Handlers et les attributs des Event Handlers conformément à la convention
* this.props.children
* getDefaultProps

C'est beaucoup! Ne vous inquiétez pas si tout est un peu flou. Bientôt, vous aurez beaucoup de pratique !

**THIS.STATE**

# **State**

Les informations dynamiques sont des informations qui peuvent changer.

Les composants React auront souvent besoin d'informations dynamiques pour effectuer le rendu. Par exemple, imaginez un composant qui affiche le score d'un match de basket. Le score du jeu peut changer avec le temps, ce qui signifie que le score est dynamique. Notre composant devra connaître le score, une information dynamique, afin de restituer de manière utile.

Il existe deux façons pour un composant d'obtenir des informations dynamiques : les props et l'état. Outre les props et l'état, chaque valeur utilisée dans un composant doit toujours rester exactement la même.

Vous venez de passer une longue leçon à apprendre les props. Il est maintenant temps d'en savoir plus sur l'état. les props et l'état sont tout ce dont vous avez besoin pour mettre en place un écosystème de composants React en interaction.

Cliquez sur Suivant pour commencer !

Par Exemple, l'état du composant de la barre de recherche inclut une valeur de terme, qui change lorsqu'un utilisateur saisit du texte dans la barre de recherche.

Question

Dans le cadre de cet exercice 16, quelles sont les différences entre props et state ?

Réponse

Il existe plusieurs différences entre les props et l'état d'un composant React, et quelques distinctions importantes sont les suivantes.

Les props sont des Data et Event Handlers qui sont transmis des composants parent aux composants enfants. Ils ne peuvent pas être modifiés une fois qu'ils sont adoptés.

L'état est similaire aux props, mais les composants ont leur propre état qu'ils stockent et contrôlent. L'état est entièrement géré au sein du composant et peut changer au fil du temps. En règle générale, l'état d'un composant est transmis à ses composants enfants et devient les props des enfants.

# **Setting Initial State**

Un composant React peut accéder aux informations dynamiques de deux manières : les props et l'état. Contrairement aux props, l'état d'un composant n'est pas transmis de l'extérieur. Un composant décide de son propre état.

Pour qu'un composant ait un state, attribuez-lui une propriété state. Cette propriété doit être déclarée à l'intérieur d'une méthode constructeur, comme ceci :

class Example extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { mood: 'decent' };  
  }  
  
  render() {  
    return <div></div>;  
  }  
}  
  
<Example />

Whoa, une méthode constructeur? super (props) ? Que se passe t-il ici? Regardons de plus près ce code potentiellement inconnu:

constructor(props) {  
  super(props);  
  this.state = { mood: 'decent' };  
}

this.state doit être égal à un objet, comme dans l'exemple ci-dessus. Cet objet représente « state » initial de toute instance de composant. Nous vous expliquerons cela plus bientôt !

Et le reste du code ? constructeur et super sont tous deux des fonctionnalités d'ES6, pas uniques à React. Il n'y a rien de particulièrement React-y dans leur utilisation ici. Une explication complète de leur fonctionnalité sort du cadre de ce cours, mais nous vous recommandons de vous familiariser. Il est important de noter que les composants React doivent toujours appeler super dans leurs constructeurs pour être configurés correctement.

Regardez en bas de l'exemple de code le plus élevé dans cette colonne. <Example /> a un state, et son state est égal à { mood: 'decent' }.

Assurez-vous de ne pas séparer le constructeur et le rendu par une virgule ! Les méthodes ne doivent jamais être séparées par des virgules si elles se trouvent à l'intérieur d'un corps de classe. Ceci pour souligner le fait que les classes et les littéraux d'objet sont différents.

### Question

In the context of this [exercise 15](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-state/exercises/initial-state), what values can you store in state?

### Answer

A React component’s state is a plain JavaScript object, so it can store any valid value that is storable within a JavaScript object. This includes booleans, strings, numbers, and even other objects which you can nest within the state.

import React from 'react';

import ReactDOM from 'react-dom';

class App extends React.Component {

  // constructor method begins here:

  constructor(props){

    super(props);

    this.state = {title : 'Best App'}

  }

  //

  render() {

    return (

      <h1>

        Wow this entire app is just an h1.

      </h1>

    );

  }

}

# **Access a Component's state**

Pour lire l'état d'un composant, utilisez l'expression this.state.name-of-property :

class TodayImFeeling extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { mood: 'decent' };  
  }  
  
  render() {  
    return (  
      <h1>  
        I'm feeling {this.state.mood}!  
      </h1>  
    );  
  }  
}

La classe de composant ci-dessus lit une propriété dans son state depuis l'intérieur de sa fonction de rendu. Tout comme this.props, vous pouvez utiliser this.state à partir de n'importe quelle propriété définie à l'intérieur du corps d'une classe de composants.

### Question

In the context of this [exercise 15](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-101-components-interacting-u/lessons/this-state/exercises/access-component-state), can we access state properties using bracket notation?

### Answer

Yes, component state is essentially a JavaScript object, so you can access the properties using dot notation as well as bracket notation.

For example,

this.state['key'];

Using bracket notation allows us to select property names that have special characters or spaces that we otherwise could not select using dot notation. For instance,

this.state = {

"key 1": value

}

/\* Not valid \*/

this.state.key 1;

/\* Valid \*/

this.state["key 1"];

# **Update state with this.setState**

Un composant peut faire plus que simplement lire son propre état. Un composant peut également changer son propre état. Un composant change d'état en appelant la fonction this.setState().

this.setState() prend deux arguments : un objet qui mettra à jour le state du composant et un callback. Vous n'avez pratiquement jamais besoin du callback.

Dans l'éditeur de code, jetez un œil à Example.js. Notez que <Exemple /> a un état de :

{  
  mood:   'great',  
  hungry: false  
}

Maintenant, disons que <Exemple /> devait appeler :

this.setState({ hungry: true });

Après cet appel, voici quel serait l'état de <Example /> :

{  
  mood:   'great',  
  hungry: true  
}

La mood de state reste inchangée !

this.setState() prend un objet et fusionne cet objet avec le state actuel du composant. S'il existe des propriétés dans le state actuel qui ne font pas partie de cet objet, ces propriétés restent telles qu'elles étaient.

# **Call this.setState from Another Function**

La façon la plus courante d'appeler this.setState() est d'appeler une fonction personnalisée qui encapsule un appel this.setState(). .makeSomeFog() est un exemple :

class Example extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { weather: 'sunny' };  
    this.makeSomeFog = this.makeSomeFog.bind(this);  
  }  
  
  makeSomeFog() {  
    this.setState({  
      weather: 'foggy'  
    });  
  }  
}

Remarquez comment la méthode makeSomeFog() contient un appel à this.setState(). Vous avez peut-être remarqué une ligne bizarre ici:

this.makeSomeFog = this.makeSomeFog.bind(this);

Cette ligne est nécessaire car le corps de makeSomeFog() contient le mot this. Nous en reparlerons plus bientôt ! Voyons comment une fonction enveloppant this.setState() pourrait fonctionner en pratique. Dans l'éditeur de code, lisez Mood.js jusqu'au bout.

Voici comment l'état d'un <Mood /> serait défini :

Un utilisateur déclenche un événement (ici un événement clic, déclenché en cliquant sur un <bouton></bouton>).

L'événement de l'étape 1 est écouté (dans ce cas par l'attribut onClick à la ligne 20).

Lorsque cet événement écouté se produit, il appelle une fonction de Event Handler (dans ce cas, this.toggleMood(), appelée à la ligne 20 et définie aux lignes 11 à 14).

À l'intérieur du corps du Event Handler, this.setState() est appelé (dans ce cas à la ligne 13).

L'état du composant est modifié !

En raison de la façon dont les Event Handlers sont liés en JavaScript, this.toggleMood() perd son this lorsqu'il est utilisé à la ligne 20. Par conséquent, les expressions this.state.mood et this.setState aux lignes 12 et 13 ne signifieront pas ce qu'ils sont censés faire… à moins que vous n'ayez déjà lié le bon this à this.toggleMood.

C'est pourquoi nous devons lier this.toggleMood à this à la ligne 8.

Pour une explication détaillée de ce type de supercherie de liaison, commencez par la documentation React. Pour les moins curieux, sachez simplement que dans React, chaque fois que vous définissez un Event Handler qui utilise this, vous devez ajouter this.methodName = this.methodName.bind(this) à votre fonction constructeur.

Regardez la déclaration à la ligne 12. Qu'est-ce que cela fait?

La ligne 12 déclare un const nommé newMood égal à l'opposé de this.state.mood. Si this.state.mood est "bon", alors newMood sera "mauvais", et vice versa.

Une instance <Mood /> afficherait "Je me sens bien !" accompagné d'un bouton. Cliquer sur le bouton changerait l'affichage en "Je me sens mal!" Cliquer à nouveau reviendrait à "Je me sens bien!", etc. Essayez de suivre la procédure pas à pas dans Mood.js et voyez comment tout cela fonctionne. Une dernière remarque : vous ne pouvez pas appeler this.setState() depuis l'intérieur de la fonction de rendu ! Nous expliquerons pourquoi dans le prochain exercice.

import React from 'react';

import ReactDOM from 'react-dom';

const green = '#39D1B4';

const yellow = '#FFD712';

class Toggle extends React.Component {

  constructor(props){

    super(props);

    this.state = { color: green };

    this.changeColor = this.changeColor.bind(this);

  }

  changeColor() {

    const newColor = this.state.color == green ? yellow : green;

    this.setState({ color: newColor });

  }

  render() {

    return (

      <div style={{background: this.state.color}}>

        <h1>

          Change my color

        </h1>

        <button onClick={this.changeColor}>

          Change color

        </button>

      </div>

    );

  }

}

ReactDOM.render(<Toggle />, document.getElementById('app'));

Vous venez d'écrire une méthode de classe de composant appelée this.setState(). Lorsque vous écrivez une méthode de classe de composant qui utilise ceci, vous devez lier cette méthode à l'intérieur de votre fonction constructeur !

Ajoutez la ligne suivante à la fin de constructor():

this.changeColor = this.changeColor.bind(this);

# **this.setState Automatically Calls render**

Il y a quelque chose d'étrange dans tout cela.

Regardez à nouveau Toggle.js.

Lorsqu'un utilisateur clique sur le <bouton></bouton>, la méthode .changeColor() est appelée. Jetez un œil à la définition de .changeColor().

.changeColor() appelle this.setState(), qui met à jour this.state.color. Cependant, même si this.state.color est mis à jour du vert au jaune, cela seul ne devrait pas suffire à faire changer la couleur de l'écran !

La couleur de l'écran ne change pas tant que Toggle n'est pas rendu.

À l'intérieur de la fonction de rendu, c'est cette ligne :

<div style={{background:this.state.color}}>

qui change la couleur d'un objet DOM virtuel en le nouveau this.state.color, provoquant éventuellement un changement dans l'écran. Si vous appelez .changeColor(), ne devriez-vous pas également appeler à nouveau .render() ? .changeColor() fait seulement en sorte que, la prochaine fois que vous effectuerez le rendu, la couleur sera différente. Pourquoi pouvez-vous voir le nouvel arrière-plan immédiatement, si vous n'avez pas rendu le composant ?

Voici pourquoi : Chaque fois que vous appelez this.setState(), this.setState() appelle AUTOMATIQUEMENT .render() dès que l'état a changé.

Pensez à this.setState() comme étant en fait deux choses : this.setState(), immédiatement suivi de .render(). C'est pourquoi vous ne pouvez pas appeler this.setState() depuis l'intérieur de la méthode .render() ! this.setState() appelle automatiquement .render(). Si .render() appelle this.setState(), une boucle infinie est créée.

# **Components Interacting Recap**

Dans cette unité, vous avez appris à utiliser import et export pour accéder à un fichier à partir d'un autre. Vous avez appris les props et le state, ainsi que les innombrables variations de leur fonctionnement.

Avant cette unité, vous avez découvert JSX, les composants et comment ils peuvent fonctionner ensemble.

Une application React n'est essentiellement qu'un grand nombre de composants, définissant l'état et se transmettant des props. Maintenant que vous avez une réelle compréhension de l'utilisation des accessoires et de le state, vous disposez de loin des outils les plus importants dont vous avez besoin !

Dans les leçons à venir, l'accent sera légèrement déplacé vers l'extérieur. En plus d'apprendre de nouvelles compétences, vous apprendrez également vos premiers modèles de programmation. Ces stratégies à grande échelle vous aideront à combiner ce que vous avez appris et à vraiment commencer à construire.

# **The Component Lifecycle**

Nous avons vu que les composants React peuvent être très dynamiques. Ils sont créés, rendus, ajoutés au DOM, mis à jour et supprimés. Toutes ces étapes font partie du cycle de vie d'un composant.

Le cycle de vie des composants comporte trois parties de haut niveau :

Mounting, lorsque le composant est initialisé et mis dans le DOM pour la première fois.

Updating, lorsque le composant est mis à jour à la suite d'un changement d'état ou d'props modifiés.

Unmounting, lorsque le composant est supprimé du DOM.

Chaque composant React avec lequel vous avez interagi fait au minimum la première étape. Si un composant n'est jamais monté, vous ne le verrez jamais !

Les composants les plus intéressants sont mis à jour à un moment donné. Un composant purement statique, comme par exemple un logo, peut ne jamais être mis à jour. Mais si l'état d'un composant change, il se met à jour. Ou si différents accessoires sont passés à un composant, il se met à jour.

Enfin, un composant est démonté lorsqu'il est supprimé du DOM. Par exemple, si vous avez un bouton qui masque un composant, il y a de fortes chances que ce composant soit démonté. Si votre application comporte plusieurs écrans, il est probable que chaque écran (et tous ses composants enfants) sera démonté. Si un composant est "actif" pendant toute la durée de vie de votre application (par exemple, un composant <App /> de niveau supérieur ou une barre de navigation persistante), il ne sera pas démonté. Mais la plupart des composants peuvent être démontés d'une manière ou d'une autre !

Il convient de noter que chaque instance de composant a son propre cycle de vie. Par exemple, si vous avez 3 boutons sur une page, il y a 3 instances de composant, chacune avec son propre cycle de vie. Cependant, une fois qu'une instance de composant est démontée, c'est tout : elle ne sera jamais remontée, ni mise à jour, ni démontée.

Instructions

Jetez un œil à ce diagramme de référence utile. Ne vous inquiétez pas, vous n'avez pas besoin de tout comprendre pour le moment, mais jetez-y un coup d'œil.

Quelles méthodes sont appelées pendant la phase de montage ? Qu'en est-il de la phase de mise à jour ?

Qu'est-ce qui provoque une mise à jour ?

Encore une fois, ce n'est pas grave si vous ne connaissez pas encore les réponses à ces questions. Nous apprendrons tout cela bientôt!

# **Introduction to Lifecycle Methods**

Les composants React ont plusieurs méthodes, appelées méthodes de cycle de vie, qui sont appelées à différentes parties du cycle de vie d'un composant. C'est ainsi que vous, le programmeur, gérez le cycle de vie d'un composant.

Vous ne le saviez peut-être pas, mais vous avez déjà utilisé deux des méthodes de cycle de vie les plus courantes : constructor() et render() ! constructor() est la première méthode appelée lors de la phase de montage. render() est appelé plus tard pendant la phase de Mounting, pour rendre le composant pour la première fois, et pendant la phase de Updating, pour re-rendre le composant.

Notez que les méthodes de cycle de vie ne correspondent pas nécessairement à une partie du cycle de vie. constructor() ne s'exécute que pendant la phase de montage, mais render() s'exécute à la fois pendant la phase de montage et de mise à jour.

Avec cette nouvelle compréhension, construisons un composant d'horloge simple.

### Question

In the context of this [exercise 11](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-102-lifecycle-methods-u/lessons/mounting-lifecycle-methods/exercises/mounting-lifecycle-methods-intro), do the mounting lifecycle methods run more than once in a component’s life?

### Answer

No, the mounting lifecycle methods (excluding render) are only run once in the entire life of a component.

This is because a component instance can only be mounted once and can never be mounted a second time. It is not possible to unmount and then “remount” a component, because once the component is unmounted it is permanently “destroyed” and cannot be mounted again.

# **componentDidMount**

Nous avons créé un composant d'horloge, mais il est statique. Ne serait-il pas agréable qu'il soit mis à jour ? À un niveau élevé, nous aimerions mettre à jour this.state.date avec une nouvelle date une fois par seconde.

JavaScript a une fonction utile, setInterval(), qui nous aidera à faire exactement cela. Il nous permet d'exécuter une fonction sur un intervalle défini. Dans notre cas, nous allons créer une fonction qui met à jour this.state.date et l'appeler toutes les secondes. Nous voudrons exécuter un code qui ressemble à ceci:

// NOTE: This code doesn't clean itself up properly.  
// We'll explore that in the next exercise.  
const oneSecond = 1000;  
setInterval(() => {  
  this.setState({ date: new Date() });  
}, oneSecond);

Nous avons le code que nous voulons exécuter, c'est génial. Mais où doit-on mettre ce code ? En d'autres termes, où dans le cycle de vie du composant doit-il aller ?

N'oubliez pas que le cycle de vie des composants comporte trois parties de haut niveau :

1. Mounting, lorsque le composant est initialisé et mis dans le DOM pour la première fois.
2. Updating, lorsque le composant est mis à jour à la suite d'un changement d'état ou d'accessoires modifiés.
3. Unmounting, lorsque le composant est supprimé du DOM.

Ce n'est certainement pas dans la phase de démontage - nous ne voulons pas commencer notre intervalle lorsque l'horloge disparaît de l'écran ! Ce n'est probablement pas non plus utile pendant la phase de mise à jour - nous voulons que l'intervalle commence dès que l'horloge apparaît, et nous ne voulons pas attendre une mise à jour. Il est probablement logique de coller ce code quelque part dans la phase de montage.

Nous avons vu deux fonctions : le render() et le constructeur. Pouvons-nous mettre ce code dans l'un ou l'autre de ces endroits ?

render() n'est pas un bon candidat. D'une part, il s'exécute pendant la phase de montage et la phase de mise à jour, trop souvent pour nous. C'est aussi généralement une mauvaise idée de configurer n'importe quel type d'effet secondaire comme celui-ci dans render(), car cela peut créer des bugs subtils à l'avenir.

constructor() n'est pas génial non plus. Il ne s'exécute que pendant la phase de montage, donc c'est bien, mais vous devriez généralement éviter les effets secondaires comme celui-ci dans les constructeurs car cela viole quelque chose appelé le principe de responsabilité unique. En bref, ce n'est pas la responsabilité d'un constructeur de déclencher des effets secondaires. (Vous pouvez en savoir plus sur le principe sur Wikipedia.)

Si ce n'est pas render() ou le constructeur, alors où ? Entrez une nouvelle méthode de cycle de vie, componentDidMount().

componentDidMount() est la dernière méthode appelée lors de la phase de montage. La commande est :

1. Le constructeur
2. render()
3. composantDidMount()

En d'autres termes, il est appelé après le rendu du composant. C'est là que nous voudrons démarrer notre minuterie. (Une autre méthode, getDerivedStateFromProps(), est appelée entre le constructeur et render(), mais elle est très rarement utilisée et n'est généralement pas la meilleure façon d'atteindre vos objectifs. Nous n'en parlerons pas dans cette leçon.)

import React from 'react';

import ReactDOM from 'react-dom';

class Clock extends React.Component {

  constructor(props) {

    super(props);

    this.state = { date: new Date() };

  }

  render() {

    return <div>{this.state.date.toLocaleTimeString()}</div>;

  }

  componentDidMount() {

    // Paste your code here.

    const oneSecond = 1000;

    setInterval(() => {

      this.setState({ date: new Date() });

    }, oneSecond);

  }

}

ReactDOM.render(<Clock />, document.getElementById('app'));

### Question

In the context of this [exercise 11](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-102-lifecycle-methods-u/lessons/mounting-lifecycle-methods/exercises/componentwillmount), when does the constructor method get called?

### Answer

The constructor method is invoked when a new component instance is created and runs before the entire component’s lifecycle starts. It runs before componentWillMount, which is the start of the mounting phase of the component.

# **componentWillUnmount**

Notre horloge fonctionne, mais elle a un problème important. Nous n'avons jamais dit à l'intervalle de s'arrêter, il continuera donc à exécuter cette fonction pour toujours (ou du moins, jusqu'à ce que l'utilisateur quitte/rafraîchisse la page).

Lorsque le composant est démonté (en d'autres termes, supprimé de la page), cette minuterie continue de fonctionner, essayant de mettre à jour l'état d'un composant qui a effectivement disparu. Cela signifie que vos utilisateurs auront du code JavaScript exécuté inutilement, ce qui nuira aux performances de votre application.

React enregistrera un avertissement qui ressemble à ceci :

Warning: Can't perform a React state update on an unmounted component. This is a no-op, but it indicates a memory leak in your application. To fix, cancel all subscriptions and asynchronous tasks in the componentWillUnmount method.

Imaginez si l'horloge est montée et démontée des centaines de fois - cela finira par ralentir votre page à cause de tout le travail inutile. Vous verrez également des avertissements dans la console de votre navigateur. Pire encore, cela peut entraîner des bogues subtils et ennuyeux.

Toutes ces mauvaises choses peuvent arriver si nous ne parvenons pas à nettoyer un effet secondaire d'un composant. Dans notre cas, il s'agit d'un appel à setInterval(), mais les composants peuvent avoir de nombreux autres effets secondaires : charger des données externes avec AJAX, modifier manuellement le DOM, définir une valeur globale, etc. Nous essayons de limiter nos effets secondaires, mais il est difficile de créer une application intéressante avec vraiment aucun effet secondaire.

En général, lorsqu'un composant produit un effet secondaire, vous devez vous rappeler de le nettoyer.

JavaScript nous donne la fonction clearInterval(). setInterval() peut renvoyer un ID, que vous pouvez ensuite passer à clearInterval() pour l'effacer. Voici le code que nous voudrons utiliser:

const oneSecond = 1000;  
this.intervalID = setInterval(() => {  
  this.setState({ date: new Date() });  
}, oneSecond);  
  
// Some time later...  
clearInterval(this.intervalID);

clearInterval(this.intervalID);

À un niveau élevé, nous voulons continuer à configurer notre setInterval() dans componentDidMount(), mais nous voulons ensuite effacer cet intervalle lorsque l'horloge est démontée.

Introduisons une nouvelle méthode de cycle de vie : componentWillUnmount(). componentWillUnmount() est appelé dans la phase de démontage, juste avant que le composant ne soit complètement détruit. C’est un moment utile pour nettoyer tout désordre de votre composant. Dans notre cas, nous l'utiliserons pour nettoyer l'intervalle de l'horloge.

import React from 'react';

export class Clock extends React.Component {

  constructor(props) {

    super(props);

    this.state = { date: new Date() };

  }

  render() {

    return <div>{ this.state.date.toLocaleTimeString() }</div>;

  }

  componentDidMount() {

    const oneSecond = 1000;

    this.intervalID = setInterval(() => {

      this.setState({ date: new Date() });

    }, oneSecond);

  }

  componentWillUnmount(){

    clearInterval(this.intervalID);

  }

}

### Question

In the context of this [exercise 1](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-102-lifecycle-methods-u/lessons/mounting-lifecycle-methods/exercises/render-mounting), is render() different for the mounting and updating lifecycle phases?

### Answer

For the mounting and updating lifecycle phases, a component will use the same render() method during both. However, the context on which they are called, and their purposes are slightly different for each phase.

During the mounting phase, render() is called exactly once, and is used to render the component for the very first time, essentially creating something that wasn’t there before in the DOM.

# During the updating phase, render() is called during every change in props and state of the component, and follows the very first render(). As a result, each render() during this phase of the component lifecycle is essentially a re-render of the component after its initial render.

# **componentDidUpdate**

N'oubliez pas les trois parties du cycle de vie d'un composant :

1. Mounting, lorsque le composant est initialisé et mis dans le DOM pour la première fois
2. Updating, lorsque le composant est mis à jour à la suite d'un changement d'état ou props modifiés
3. Unmounting, lorsque le composant est supprimé du DOM

We’ve looked at mounting (constructor(), render(), and componentDidMount()). We’ve looked at unmounting (componentWillUnmount()). Let’s finish by looking at the updating phase.

Une mise à jour est causée par des modifications apportées aux props ou à state. Vous avez déjà vu cela se produire un tas de fois. Chaque fois que vous avez appelé setState() avec de nouvelles données, vous avez déclenché une mise à jour. Chaque fois que vous modifiez les props passés à un composant, vous l'avez mis à jour.

Lorsqu'un composant se met à jour, il appelle plusieurs méthodes, mais seules deux sont couramment utilisées.

Le premier est render(), que nous avons vu dans chaque composant React. Lorsque les props ou state d'un composant changent, render() est appelé.

Le second, que nous n'avons pas encore vu, est componentDidUpdate(). Tout comme componentDidMount() est un bon endroit pour la configuration de la phase de Mounting, componentDidUpdate() est un bon endroit pour le travail de la phase de Updating.

Question

In the context of this [exercise 11](https://www.codecademy.com/paths/web-development/tracks/front-end-applications-with-react/modules/react-102-lifecycle-methods-u/lessons/mounting-lifecycle-methods/exercises/componentdidmount), introducing the componentDidMount() method, can you set the component’s state in this method?

Answer

Yes, you can set the component’s state in this method, using the setState() method. This is not possible in some other lifecycle methods, such as render(), because calling setState() within them can result in infinite loops.

Calling setState() in componentDidMount() will not end up in an infinite loop, but will just invoke an additional rendering. This additional rendering will not be visible as both of the renders will happen be right before the browser updates the screen, but, one drawback to doing this is that the additional render can take some additional time and cause the site to display slower.

Instead of setting the state initially within componentDidMount(), you may consider setting the initial state in the constructor() method instead.

# **Review**

Nous sommes arrivés à la fin de la leçon. Nous avons découvert les trois phases principales du cycle de vie d'un composant :

Mounting, lorsque le composant est initialisé et mis dans le DOM pour la première fois. Nous avons vu que le constructeur, render() et componentDidMount() sont appelés pendant cette phase.

Updating, lorsque le composant est mis à jour à la suite d'un changement d'état ou de props modifiés. Nous avons vu que render() et componentDidUpdate() sont appelés pendant cette phase.

Unmouting, lorsque le composant est supprimé du DOM. Nous avons vu que componentWillUnmount() était appelé ici, ce qui était un bon moment pour nettoyer les choses.

Nous avons également appris à créer des effets secondaires et à les supprimer. Nous savons désormais fabriquer des composants plus robustes et plus complexes !

À votre droite se trouve une référence que vous pouvez utiliser. Il montre les trois phases du cycle de vie d'un composant et quelles méthodes sont appelées à chaque phase. Vous pouvez également vous référer à ce schéma interactif.

Pour en savoir plus, vous pouvez lire la documentation officielle de React. Voir « État et cycle de vie » et la documentation de React.Component.