CHAPTER 6 :CONDITIONAL RENDERING

Your components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like if statements, &&, and ? : operators.
How to return different JSX depending on a condition
How to conditionally include or exclude a piece of JSX
Common conditional syntax shortcuts you’ll encounter in React codebases

Conditionally returning JSX

Let’s say you have a PackingList component rendering several Items, which can be marked as packed or not:

```
function Item({ name, isPacked })
{
  return <li
className="item">{name}</li>
;
}

export default function
PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing
List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a
golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
```

Sally Ride's Packing List

- Space suit
- Helmet with a golden leaf
- Photo of Tam

```
    </ul>
  </section>
 );
}
```

Notice that some of the Item components have their isPacked prop set to true instead of false. You want to add a checkmark (   ) to packed items if isPacked={true}.

You can write this as an if/else statement like so:

```
if (isPacked) {
  return <li className="item">{name}     </li>;
}
return <li className="item">{name}</li>;
```

Sally Ride's Packing List

- Space suit
- Helmet with a golden leaf
- Photo of Tam

Try editing what gets returned in either case, and see how the result changes!

Notice how you're creating branching logic with JavaScript's if and return statements. In React, control flow (like conditions) is handled by JavaScript.

Conditionally returning nothing with null

In some situations, you won't want to render anything at all. For example, say you don't want to show packed items at all. A component must return something. In this case, you can return null:

```
if (isPacked) {

  return null;

}

return <li className="item">{name}</li>;
```

If isPacked is true, the component will return nothing, null. Otherwise, it will return JSX to render.

```
function Item({ name,
isPacked }) {
  if (isPacked) {
    return null;
  }
  return <li
className="item">{name
}</li>;
}
export default function
PackingList() {
  return (
   section>
      <h1>Sally Ride's
Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
```

Sally Ride's Packing List

- **Photo of Tam**

```
          name="Helmet with
a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of
Tam"
        />
      </ul>
    </section>  );}
```

In practice, returning null from a component isn't common because it might surprise a developer trying to render it. More often, you would conditionally include or exclude the component in the parent component's JSX. Here's how to do that!

Conditionally including JSX

In the previous example, you controlled which (if any!) JSX tree would be returned by the component. You may already have noticed some duplication in the render output:

```
<li className="item">{name}     </li>
```

is very similar to

```
<li className="item">{name}</li>
```

Both of the conditional branches return <li className="item">...</li>:

```
if (isPacked) {
  return <li className="item">{name}     </li>;
}
return <li className="item">{name}</li>;
```

While this duplication isn’t harmful, it could make your code harder to maintain. What if you want to change the className? You’d have to do it in two places in your code! In such a situation, you could conditionally include a little JSX to make your code more DRY.

Conditional (ternary) operator (? :)

JavaScript has a compact syntax for writing a conditional expression — the conditional operator or “ternary operator”.

Instead of this:

```
if (isPacked) {
  return <li className="item">{name}     </li>;
}
return <li className="item">{name}</li>;
```

You can write this:

```
return (
  <li className="item">
    {isPacked ? name + '     ' : name}
  </li>
);
```

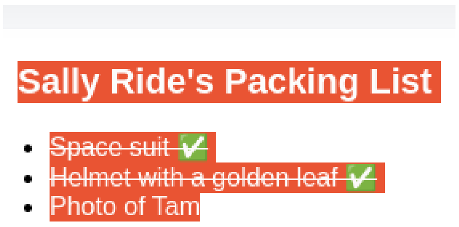
You can read it as “if isPacked is true, then (?) render name + '   ', otherwise (:) render name”.

If you’re coming from an object-oriented programming background, you might assume that the two examples above are subtly different because one of them may create two different “instances” of <li>. But JSX elements aren’t “instances” because they don’t hold any internal state and aren’t real DOM nodes. They’re lightweight descriptions, like blueprints. So these two examples, in fact, are completely equivalent. Preserving and Resetting State goes into detail about how this works.

Now let’s say you want to wrap the completed item’s text into another HTML tag, like <del> to strike it out. You can add even more newlines and parentheses so that it’s easier to nest more JSX in each of the cases:

```
function Item({ name,
isPacked }) {
  return (
    <li className="item">
      {isPacked ? (
        <del>
          {name + '    '}
        </del>
      ) : (
        name
      )}
    </li>
  );
}

export default function
PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing
List</h1>
      <ul>
        <Item
```

```
          isPacked={true}
```



Sally Ride's Packing List

- ~~Space suit ✅~~
- ~~Helmet with a golden leaf ✅~~
- Photo of Tam

```
name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a
golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

A JavaScript && expression returns the value of its right side (in our case, the checkmark) if the left side (our condition) is true. But if the condition is false, the whole expression becomes false. React considers false as a “hole” in the JSX tree, just like null or undefined, and doesn’t render anything in its place.

Conditionally assigning JSX to a variable

When the shortcuts get in the way of writing plain code, try using an if statement and a variable. You can reassign variables defined with let, so start by providing the default content you want to display, the name:

```
let itemContent = name;
```

Use an if statement to reassign a JSX expression to itemContent if isPacked is true:

```
if (isPacked) {
  itemContent = name + "     ";
}
```

Curly braces open the “window into JavaScript”. Embed the variable with curly braces in the returned JSX tree, nesting the previously calculated expression inside of JSX:

```
<li className="item">
  {itemContent}
</li>
```

This style is the most verbose, but it’s also the most flexible. Here it is in action:

```
function Item({ name, isPacked }) {
  let itemContent = name;
  if (isPacked) {
    itemContent = name + "     ";
  }
  return (
    <li className="item">
      {itemContent}
    </li>
  );
}
export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

Sally Ride's Packing List

- Space suit
- Helmet with a golden leaf
- Photo of Tam

Like before, this works not only for text, but for arbitrary JSX too:

```
function Item({ name, isPacked }) {
  let itemContent = name;
  if (isPacked) {
    itemContent = (
      <del>
        {name + "     "}
      </del>
    );
  }
  return (
    <li className="item">
      {itemContent}
    </li>
  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
        isPacked={true}
        name="Space suit"
      />
      <Item
        isPacked={true}
        name="Helmet with a golden leaf"
      />
      <Item
        isPacked={false}
        name="Photo of Tam"
      />
    </ul>
  </section>
 );
}
```

CHAPTER 7 : Rendering Lists

You will often want to display multiple similar components from a collection of data. You can use the JavaScript array methods to manipulate an array of data. On this page, you'll use filter() and

() with React to filter and transform your array of data into an array of components.

7.1 How to render components from an array using JavaScript's map()

7.2 How to render only specific components using JavaScript’s filter()

7.3 When and why to use React keys

Rendering data from arrays

Say that you have a list of content.

```
<ul>
  <li>Creola Katherine Johnson: mathematician</li>
  <li>Mario José Molina-Pasquel Henríquez: chemist</li>
  <li>Mohammad Abdus Salam: physicist</li>
  <li>Percy Lavon Julian: chemist</li>
  <li>Subrahmanyan Chandrasekhar: astrophysicist</li>
</ul>
```

The only difference among those list items is their contents, their data. You will often need to show several instances of the same component using different data when building interfaces: from lists of comments to galleries of profile images. In these situations, you can store that data in JavaScript objects and arrays and use methods like map() and filter() to render lists of components from them.

Here’s a short example of how to generate a list of items from an array:

1.Move the data into an array:

```
const people = [
  'Creola Katherine Johnson: mathematician',
  'Mario José Molina-Pasquel Henríquez: chemist',
  'Mohammad Abdus Salam: physicist',
  'Percy Lavon Julian: chemist',
  'Subrahmanyan Chandrasekhar: astrophysicist'
];
```

2.Map the people members into a new array of JSX nodes, listItems:

```
const listItems = people.map(person => <li>{person}</li>);
```

3.Return listItems from your component wrapped in a <ul>:

```
return <ul>{listItems}</ul>;
```

```
const people = [
  'Creola Katherine Johnson: mathematician',
  'Mario José Molina-Pasquel Henríquez: chemist',
  'Mohammad Abdus Salam: physicist',
  'Percy Lavon Julian: chemist',
  'Subrahmanyan Chandrasekhar: astrophysicist'
];

export default function List() {
  const listItems = people.map(person =>
    <li>{person}</li>
  );
  return <ul>{listItems}</ul>;
}
```

Creola Katherine Johnson: mathematician

- Mario José Molina-Pasquel Henríquez: chemist
- Mohammad Abdus Salam: physicist
- Percy Lavon Julian: chemist
- Subrahmanyan Chandrasekhar: astrophysicist

Warning: Each child in a list should have a unique “key” prop.

```
const people = [{
  id: 0,
  name: 'Creola Katherine Johnson',
  profession: 'mathematician',
}, {
  id: 1,
  name: 'Mario José Molina-Pasquel Henríquez',
  profession: 'chemist',
}, {
  id: 2,
  name: 'Mohammad Abdus Salam',
  profession: 'physicist',
}, {
  id: 3,
  name: 'Percy Lavon Julian',
  profession: 'chemist',
}, {
  id: 4,
  name: 'Subrahmanyan Chandrasekhar',
  profession: 'astrophysicist',
}];
```

Let’s say you want a way to only show people whose profession is 'chemist'. You can use JavaScript’s filter() method to return just those people. This method takes an array of items, passes them through a “test” (a function that returns true or false), and returns a new array of only those items that passed the test (returned true).

You only want the items where profession is 'chemist'. The “test” function for this looks like (person) =>

person.profession === 'chemist'. Here’s how to put it together:

Create a new array of just “chemist” people, chemists, by calling filter() on the people filtering by person.profession === 'chemist':

```
const chemists = people.filter(person =>
  person.profession === 'chemist'
);
```

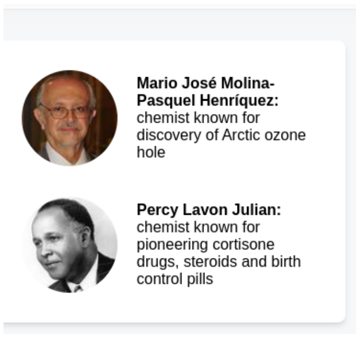
Now map over chemists:

```
const listItems = chemists.map(person =>
  <li>
     <img
       src={getImageUrl(person)}
       alt={person.name}
     />
     <p>
       <b>{person.name}:</b>
       {' ' + person.profession + ' '}
       known for {person.accomplishment}
     </p>
  </li>
);
```

Lastly, return the listItems from your component:

```
return <ul>{listItems}</ul>;
```

```
import { people } from './data.js';
import { getImageUrl } from
'./utils.js';

export default function List() {
  const chemists =
people.filter(person =>
    person.profession ===
'chemist'
  );
  const listItems =
chemists.map(person =>
    <li>
      <img
        src={getImageUrl(person)}
        alt={person.name}
      />
      <p>
        <b>{person.name}:</b>
        {' ' + person.profession + '
'}
        known for
{person.accomplishment}
      </p>
    </li>
  );
  return <ul>{listItems}</ul>;
}
```



Mario José Molina-Pasquel Henríquez: chemist known for discovery of Arctic ozone hole

Percy Lavon Julian: chemist known for pioneering cortisone drugs, steroids and birth control pills

Arrow functions implicitly return the expression right after =>, so you didn’t need a return statement:

```
const listItems = chemists.map(person =>
  <li>...</li> // Implicit return!
);
```

However, you must write return explicitly if your => is followed by a { curly brace!

```
const listItems = chemists.map(person => { // Curly brace
  return <li>...</li>;
});
```

Arrow functions containing => { are said to have a "block body". They let you write more than a single line of code, but you have to write a return statement yourself. If you forget it, nothing gets returned!

Keeping list items in order with key

Notice that all the sandboxes above show an error in the console:

You need to give each array item a key — a string or a number that uniquely identifies it among other items in that array:

```
<li key={person.id}>...</li>
```

JSX elements directly inside a map() call always need keys!

Keys tell React which array item each component corresponds to, so that it can match them up later. This becomes important if your array items can move (e.g. due to sorting), get inserted, or get deleted. A well-chosen key helps React infer what exactly has happened, and make the correct updates to the DOM tree.

Rather than generating keys on the fly, you should include them in your data:

//app.js
import { people } from './data.js';
import { getImageUrl } from './utils.js';
 	
export default function List() {
  const listItems = people.map(person =>
    <li key={person.id}>
	//data.js'
	export const people = [{
	  id: 0, // Used in JSX as a key
	  name: 'Creola Katherine Johnson',
	  profession: 'mathematician',
	  accomplishment: 'spaceflight calculations',
	  imageId: 'MK3eW3A'

```
      <img
        src={getImageUrl(person)}
        alt={person.name}
      />
      <p>
        <b>{person.name}</b>
          {' ' + person.profession + ' 
'}
          known for 
{person.accomplishment}
      </p>
    </li>
  );
  return <ul>{listItems}</ul>;
}
```

```
}, {
  id: 1, // Used in JSX as a key
  name: 'Mario José Molina-
Pasquel Henríquez',
  profession: 'chemist',
  accomplishment: 'discovery of 
Arctic ozone hole',
  imageId: 'mynHUSa'
}, {
  id: 2, // Used in JSX as a key
  name: 'Mohammad Abdus 
Salam',
  profession: 'physicist',
  accomplishment: 
'electromagnetism theory',
  imageId: 'bE7W1ji'
}, {
  id: 3, // Used in JSX as a key
  name: 'Percy Lavon Julian',
  profession: 'chemist',
  accomplishment: 'pioneering 
cortisone drugs, steroids and 
birth control pills',
  imageId: 'IOjWm71'
}, {
  id: 4, // Used in JSX as a key
  name: 'Subrahmanyan 
Chandrasekhar',
  profession: 'astrophysicist',
  accomplishment: 'white dwarf 
star mass calculations',
  imageId: 'lrWQx8l'
}];
```

```
//util.js
export function 
getImageUrl(person) {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    's.jpg'
  );
}
```

What do you do when each item needs to render not one, but several DOM nodes?

The short <>...</> Fragment syntax won’t let you pass a key, so you need to either group them into a single <div>, or use the slightly longer and more explicit <Fragment> syntax:

```
import { Fragment } from 'react';

// ...

const listItems = people.map(person =>
  <Fragment key={person.id}>
    <h1>{person.name}</h1>
    <p>{person.bio}</p>
  </Fragment>
);
```

Where to get your key

Different sources of data provide different sources of keys:

Data from a database: If your data is coming from a database, you can use the database keys/IDs, which are unique by nature.

Locally generated data: If your data is generated and persisted locally (e.g. notes in a note-taking app), use an incrementing counter, crypto.randomUUID() or a package like uuid when creating items**.**

Rules of keys

Keys must be unique among siblings. However, it’s okay to use the same keys for JSX nodes in different arrays.

Keys must not change or that defeats their purpose! Don’t generate them while rendering.

Why does React need keys?

Imagine that files on your desktop didn’t have names. Instead, you’d refer to them by their order — the first file, the second file, and so on. You could get used to it, but once you delete a file, it would get confusing. The second file would become the first file, the third file would be the second file, and so on.

File names in a folder and JSX keys in an array serve a similar purpose. They let us uniquely identify an item between its siblings. A well-chosen key provides more information than the position within the array. Even if the position changes due to reordering, the key lets React identify the item throughout its lifetime.

CHAPTER 8:

Keeping Components Pure

Some JavaScript functions are pure. Pure functions only perform a calculation and nothing more. By strictly only writing your components as pure functions, you can avoid an entire class of baffling bugs and unpredictable behavior as your codebase grows. To get these benefits, though, there are a few rules you must follow.

You will learn

What purity is and how it helps you avoid bugs

How to keep components pure by keeping changes out of the render phase

How to use Strict Mode to find mistakes in your components

Purity: Components as formulas

In computer science (and especially the world of functional programming), a pure function is a function with the following characteristics:

It minds its own business. It does not change any objects or variables that existed before it was called.

Same inputs, same output. Given the same inputs, a pure function should always return the same result.

You might already be familiar with one example of pure functions: formulas in math.

Consider this math formula: y = 2x.

If x = 2 then y = 4. Always.

If x = 3 then y = 6. Always.

If x = 3, y won’t sometimes be 9 or –1 or 2.5 depending on the time of day or the state of the stock market.

If y = 2x and x = 3, y will always be 6.

If we made this into a JavaScript function, it would look like this:

```
function double(number) {
  return 2 * number;
}
```

In the above example, double is a pure function. If you pass it 3, it will return 6. Always.

React is designed around this concept. React assumes that every component you write is a pure function. This means that React components you write must always return the same JSX given the same inputs:

App.js

```
function Recipe({ drinkers }) {
  return (
    <ol>
      <li>Boil {drinkers} cups of water.</li>
      <li>Add {drinkers} spoons of tea and {0.5 * drinkers} spoons of spice.</li>
      <li>Add {0.5 * drinkers} cups of milk to boil and sugar to taste.</li>
    </ol>
  );
}

export default function App() {
  return (
    <section>
      <h1>Spiced Chai Recipe</h1>
      <h2>For two</h2>
      <Recipe drinkers={2} />
      <h2>For a gathering</h2>
      <Recipe drinkers={4} />
    </section>
  );
}
```

When you pass drinkers={2} to Recipe, it will return JSX containing 2 cups of water. Always.

If you pass drinkers={4}, it will return JSX containing 4 cups of water. Always.

Just like a math formula.

You could think of your components as recipes: if you follow them and don’t introduce new ingredients during the cooking process, you will get the same dish every time. That “dish” is the JSX that the component serves to React to render.

A tea recipe for x people: take x cups of water, add x spoons of tea and 0.5x spoons of spices, and 0.5x cups of milk

Illustrated by Rachel Lee Nabors

Side Effects: (un)intended consequences

React’s rendering process must always be pure. Components should only return their JSX, and not change

any objects or variables that existed before rendering—that would make them impure!

Here is a component that breaks this rule:

App.js

```
let guest = 0;

function Cup() {

  // Bad: changing a preexisting variable!

  guest = guest + 1;

  return <h2>Tea cup for guest #{guest}</h2>;

}

export default function TeaSet() {

  return (

    <>

      <Cup />

      <Cup />

      <Cup />

    </>

  );

}
```

###Corrected PURE one

App.js

```
function Cup({ guest }) {

  return <h2>Tea cup for guest #{guest}</h2>;

}


export default function TeaSet() {

  return (

    <>

      <Cup guest={1} />

      <Cup guest={2} />

      <Cup guest={3} />

    </>

  );

}
```

This component is reading and writing a guest variable declared outside of it. This means that calling this

component multiple times will produce different JSX! And what’s more, if other components read guest, they will produce different JSX, too, depending on when they were rendered! That’s not predictable.

Going back to our formula y = 2x, now even if x = 2, we cannot trust that y = 4. Our tests could fail, our users would be baffled, planes would fall out of the sky—you can see how this would lead to confusing bugs!

You can fix this component by passing guest as a prop instead:

Now your component is pure, as the JSX it returns only depends on the guest prop.

In general, you should not expect your components to be rendered in any particular order. It doesn’t matter if you call y = 2x before or after y = 5x: both formulas will resolve independently of each other. In the same way, each component should only “think for itself”, and not attempt to coordinate with or depend upon others during rendering. Rendering is like a school exam: each component should calculate JSX on their own!

Deep Dive

Detecting impure calculations with StrictMode

Although you might not have used them all yet, in React there are three kinds of inputs that you can read while rendering: props, state, and context. You should always treat these inputs as read-only.

When you want to change something in response to user input, you should set state instead of writing to a variable. You should never change preexisting variables or objects while your component is rendering.

React offers a “Strict Mode” in which it calls each component’s function twice during development. By calling the component functions twice, Strict Mode helps find components that break these rules.

Notice how the original example displayed “Guest #2”, “Guest #4”, and “Guest #6” instead of “Guest #1”, “Guest #2”, and “Guest #3”. The original function was impure, so calling it twice broke it. But the fixed pure version works even if the function is called twice every time. Pure functions only calculate, so calling them twice won’t change anything—just like calling double(2) twice doesn’t change what’s returned, and solving y = 2x twice doesn’t change what y is. Same inputs, same outputs. Always.

Strict Mode has no effect in production, so it won’t slow down the app for your users. To opt into Strict Mode, you can wrap your root component into <React.StrictMode>. Some frameworks do this by default.

Local mutation: Your component’s little secret

In the above example, the problem was that the component changed a preexisting variable while rendering. This is often called a “mutation” to make it sound a bit scarier. Pure functions don’t mutate variables outside of the function’s scope or objects that were created before the call—that makes them impure!

However, it’s completely fine to change variables and objects that you’ve just created while rendering. In this example, you create an [] array, assign it to a cups variable, and then push a dozen cups into it:

App.js

```
function Cup({ guest }) {
  return <h2>Tea cup for guest #{guest}</h2>;
}
export default function TeaGathering() {
  let cups = [];
  for (let i = 1; i <= 12; i++) {
    cups.push(<Cup key={i} guest={i} />);
  }
  return cups;
}
```

If the cups variable or the [] array were created outside the TeaGathering function, this would be a huge problem! You would be changing a preexisting object by pushing items into that array.

However, it’s fine because you’ve created them during the same render, inside TeaGathering. No code outside of TeaGathering will ever know that this happened. This is called “local mutation”—it’s like your component’s little secret.

Where you can cause side effects

While functional programming relies heavily on purity, at some point, somewhere, something has to change. That’s kind of the point of programming! These changes—updating the screen, starting an animation, changing the data—are called side effects. They’re things that happen “on the side”, not during rendering.

In React, side effects usually belong inside event handlers. Event handlers are functions that React runs when you perform some action—for example, when you click a button. Even though event handlers are defined inside your component, they don’t run during rendering! So event handlers don’t need to be pure.

If you’ve exhausted all other options and can’t find the right event handler for your side effect, you can still attach it to your returned JSX with a useEffect call in your component. This tells React to execute it later, after rendering, when side effects are allowed. However, this approach should be your last resort.

When possible, try to express your logic with rendering alone. You’ll be surprised how far this can take you!

Deep Dive

Why does React care about purity?

Writing pure functions takes some habit and discipline. But it also unlocks marvelous opportunities:

Your components could run in a different environment—for example, on the server! Since they return the same result for the same inputs, one component can serve many user requests.

You can improve performance by skipping rendering components whose inputs have not changed. This is safe because pure functions always return the same results, so they are safe to cache.

If some data changes in the middle of rendering a deep component tree, React can restart rendering without wasting time to finish the outdated render. Purity makes it safe to stop calculating at any time.

Every new React feature we’re building takes advantage of purity. From data fetching to animations to performance, keeping components pure unlocks the power of the React paradigm.

Recap

A component must be pure, meaning:

It minds its own business. It should not change any objects or variables that existed before rendering.

Same inputs, same output. Given the same inputs, a component should always return the same JSX.

Rendering can happen at any time, so components should not depend on each others’ rendering sequence.

You should not mutate any of the inputs that your components use for rendering. That includes props, state, and context. To update the screen, “set” state instead of mutating preexisting objects.

Strive to express your component’s logic in the JSX you return. When you need to “change things”, you’ll usually want to do it in an event handler. As a last resort, you can useEffect.

Writing pure functions takes a bit of practice, but it unlocks the power of React’s paradigm.

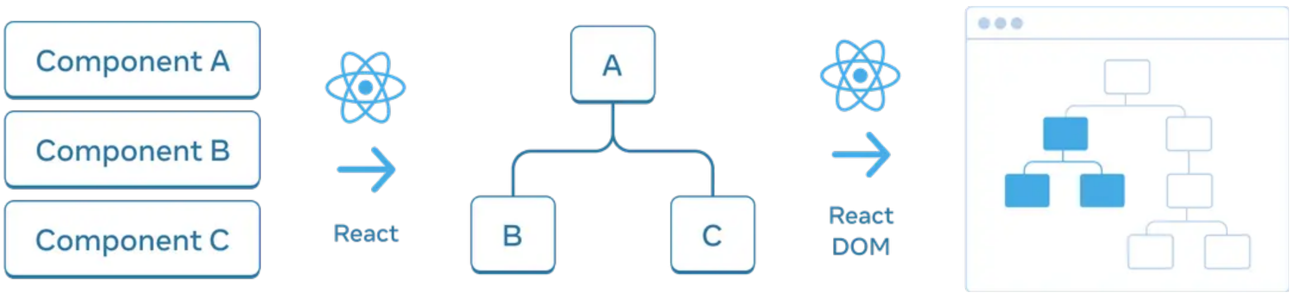
CHAPTER 9: Understanding Your UI as a Tree

Your React app is taking shape with many components being nested within each other. How does React keep track of your app's component structure?

React, and many other UI libraries, model UI as a tree. Thinking of your app as a tree is useful for understanding the relationship between components. This understanding will help you debug future concepts like performance and state management.

UI as a tree

Trees are a relationship model between items and UI is often represented using tree structures. For example, browsers use tree structures to model HTML (DOM) and CSS (CSSOM). Mobile platforms also use trees to represent their view hierarchy.



Like browsers and mobile platforms, React also uses tree structures to manage and model the relationship between components in a React app. These trees are useful tools to understand how data flows through a React app and how to optimize rendering and app size.

The Render Tree

A major feature of components is the ability to compose components of other components. As we nest components, we have the concept of parent and child components, where each parent component may itself be a child of another component.

When we render a React app, we can model this relationship in a tree, known as the render tree.

Here is a React app that renders inspirational quotes.

```
import FancyText from
'./FancyText';
import InspirationGenerator from
'./InspirationGenerator';
import Copyright from
'./Copyright';

export default function App() {
  return (
    <>
      <FancyText title text="Get
Inspired App" />
      <InspirationGenerator>
        <Copyright year={2004} />
      </InspirationGenerator>
    </>
  );
}
```

```
export default function
FancyText({title, text}) {
  return title
    ? <h1 className='fancy
title'>{text}</h1>
    : <h3 className='fancy
cursive'>{text}</h3>
}
```

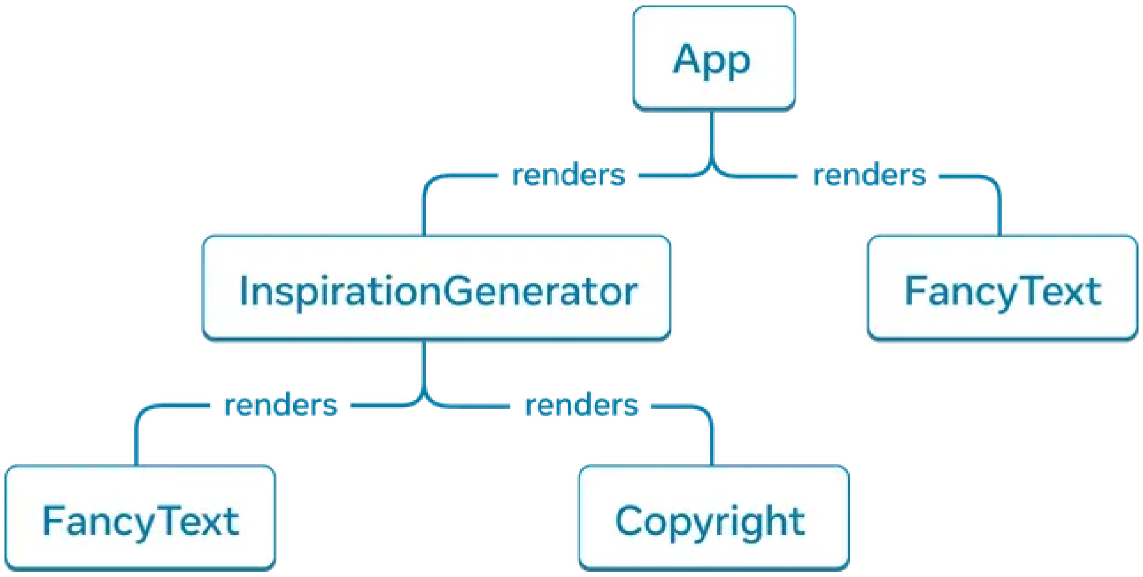
```
import * as React from 'react';
import quotes from './quotes';
import FancyText from
'./FancyText';

export default function
```

```
export default function
Copyright({year}) {
  return <p className='small'>©
{year}</p>;
}
```

```
InspirationGenerator({children})
{
  const [index, setIndex] =
React.useState(0);
  const quote = quotes[index];
  const next = () =>
setIndex((index + 1) %
quotes.length);

  return (
    <>
      <p>Your inspirational quote
is:</p>
      <FancyText text={quote} />
      <button
onClick={next}>Inspire me
again</button>
      {children}
    </>
  );
}
export default [
  "Don’t let yesterday take up too
much of today.” — Will Rogers",
  "Ambition is putting a ladder
against the sky.",
  "A joy that's shared is a joy
made double.",
  ];
```

From the example app, we can construct the above render tree.

The
tree is



composed of nodes, each of which represents a component. App, FancyText, Copyright, to name a few, are all nodes in our tree.

The root node in a React render tree is the root component of the app. In this case, the root component is App and it is the first component React renders. Each arrow in the tree points from a parent component to a child component.

A render tree represents a single render pass of a React application. With conditional rendering, a parent component may render different children depending on the data passed.

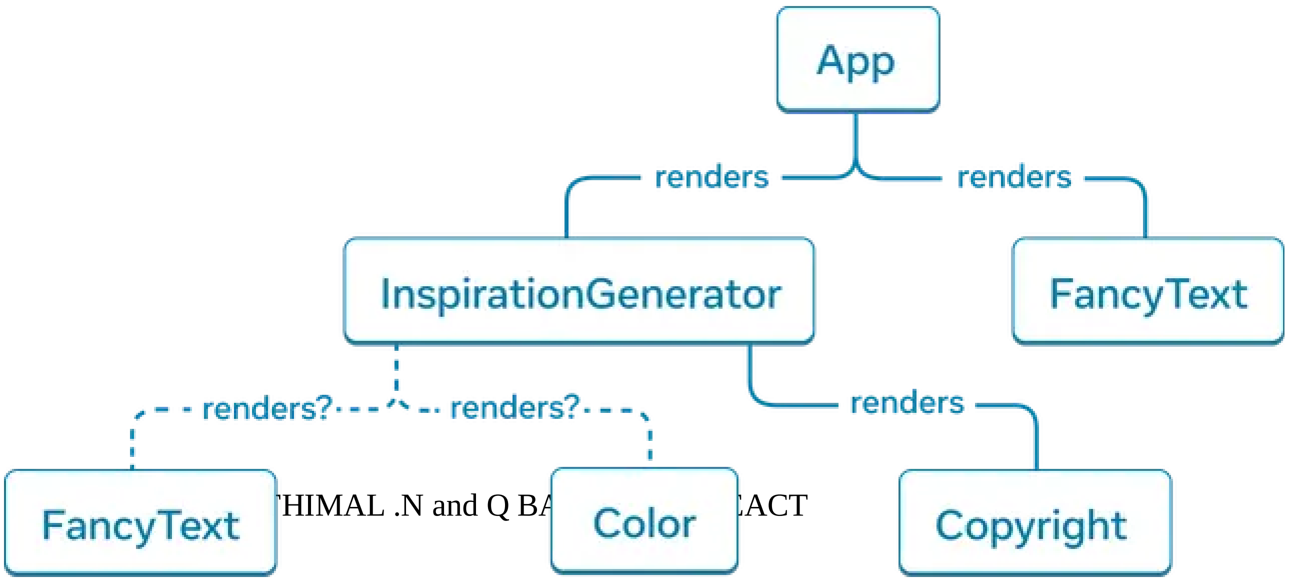
We can update the app to conditionally render either an inspirational quote or color.

```
import FancyText from './FancyText';

import InspirationGenerator from './InspirationGenerator';

import Copyright from './Copyright';


export default function App() {
  return (
    <>
      <FancyText title text="Get Inspired App" />
      <InspirationGenerator>
        <Copyright year={2004} />
      </InspirationGenerator>
    </>
  );
}
```

In this example, depending on what inspiration.type is, we may render <FancyText> or <Color>. The render tree may be different for each render pass.

Although render trees may differ across render passes, these trees are generally helpful for identifying what the *top-level* and *leaf components* are in a React app. Top-level components are the components nearest to the root component and affect the rendering performance of all the components beneath them and often contain the most complexity. Leaf components are near the bottom of the tree and have no child components and are often frequently re-rendered.

Identifying these categories of components are useful for understanding data flow and performance of your app.



In this example, depending on what inspiration.type is, we may render <FancyText> or <Color>. The render tree may be different for each render pass.

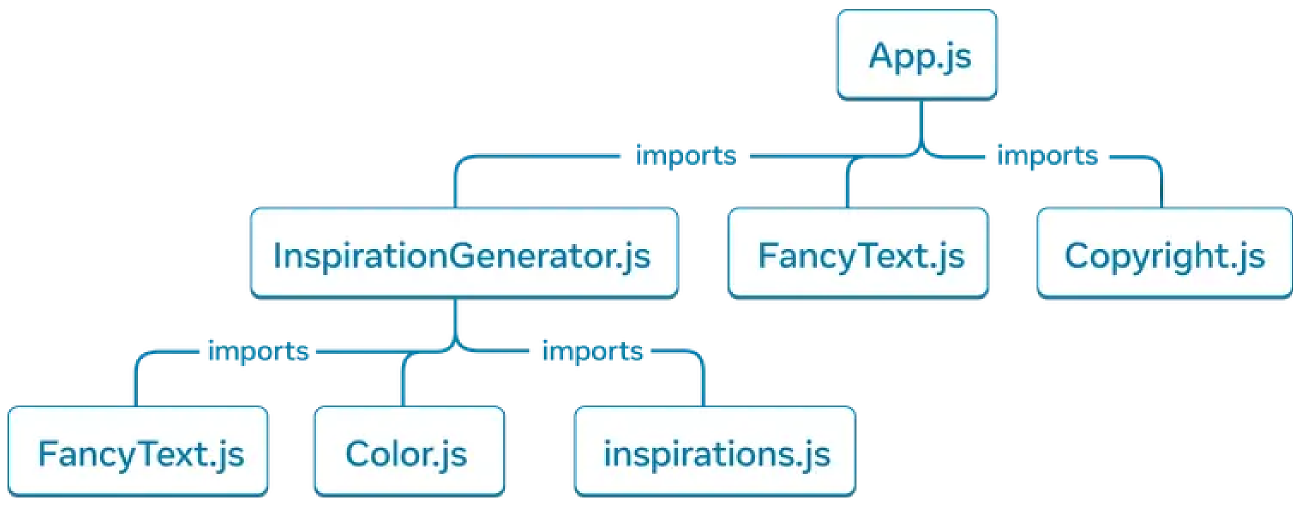
Although render trees may differ across render passes, these trees are generally helpful for identifying what the top-level and leaf components are in a React app. Top-level components are the components nearest to the root component and affect the rendering performance of all the components beneath them and often contain the most complexity. Leaf components are near the bottom of the tree and have no child components and are often frequently re-rendered.

Identifying these categories of components are useful for understanding data flow and performance of your app.

The Module Dependency Tree

Another relationship in a React app that can be modeled with a tree are an app’s module dependencies. As we break up our components and logic into separate files, we create JS modules where we may export components, functions, or constants.

Each node in a module dependency tree is a module and each branch represents an import statement in that module.If we take the previous Inspirations app, we can build a module dependency tree, or dependency tree for short.



The root node of the tree is the root module, also known as the entrypoint file. It often is the module that contains the root component.

Comparing to the render tree of the same app, there are similar structures but some notable differences:

The nodes that make-up the tree represent modules, not components.

Non-component modules, like inspirations.js, are also represented in this tree. The render tree only encapsulates components.

Copyright.js appears under App.js but in the render tree, Copyright, the component, appears as a child of InspirationGenerator. This is because InspirationGenerator accepts JSX as children props, so it renders Copyright as a child component but does not import the module.

Dependency trees are useful to determine what modules are necessary to run your React app. When building a React app for production, there is typically a build step that will bundle all the necessary JavaScript to ship to the client. The tool responsible for this is called a bundler, and bundlers will use the dependency tree to determine what modules should be included.

As your app grows, often the bundle size does too. Large bundle sizes are expensive for a client to download and run. Large bundle sizes can delay the time for your UI to get drawn. Getting a sense of your app’s dependency tree may help with debugging these issues