The React Conditionals Cheatsheet

1. Use if-statements primarily. No need for else or else-if.

Let's start with the most basic type of conditional in React. If we have data, we want to display it. If not, we want to show nothing.

Simple! How would we write that?

Let's say we are fetching an array of posts data from an API. When it is fetching the data, `posts` has a value of `undefined`.

We can check for that value with a simple if-statement.

```
export default function App() {
  const { posts } = usePosts();// posts === undefined at firstif (!posts) return null;

  return (
    <div>
      <PostList posts={posts} />
    </div>
  );
}
```

The reason this patterns works is that we are returning early. If the condition is met (if `!posts` is has a boolean value of `true`), we display nothing in our component by returning `null`.

If statements also work when you have multiple conditions that you want to check for.

For example, if you want to check for loading and error states before you display your data:

```
export default function App() {
  const { isLoading, isError, posts } = usePosts();

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>Error!</div>;
  return (
    <div>
      <PostList posts={posts} />
    </div>
  );
}
```

Notice that we can reuse the if-statement and do not have to write if-else or if-else-if, which cuts down on the code that we have to write and is still just as readable.

2. Use the ternary operator to write conditionals in your JSX

If-statements are great when we want to exit early and display nothing or a totally different component.

However, what if we don't want to write a conditional separate from our returned JSX, but directly within it?

In React, we must include expressions (something that resolves to a value), not statements within our JSX.

This is why we must write conditionals in our JSX only with ternaries and not if-statements.

For example, if we wanted to display one nested component on a mobile-sized screen and another on a larger screen, a ternary would be a perfect choice:

```
function App() {
  const isMobile = useWindowSize()

  return (
    <main>
      <Header />
      <Sidebar />
      {isMobile ? <MobileChat /> : <Chat />}
    </main>
  )
}
```

Most developers think this the only pattern they can leverage when it comes to using ternaries.

In fact, you don't have to clutter your component tree by including all of these ternaries directly in your returned JSX.

Since ternaries resolve to a value, remember that you can assign the result of a ternary to a variable, which you can then use where you like:

```
function App() {
  const isMobile = useWindowSize();

  const ChatComponent = isMobile ? <MobileChat /> : <Chat />;

  return (
    <main>
      <Header />
      <Sidebar />
      {ChatComponent}
    </main>
  )
}
```

3. No else condition? Use the && (and) operator

In many cases, you will want to use a ternary in your JSX, but will realize that if that condition is not met, you don't want to display anything.

This ternary would look like the following: `condition ? <Component /> : null`.

We

If you don't have an else condition, use the && operator

```
export default function PostFeed() {
  const { posts, hasFinished } = usePosts()

  return (
    <>
      <PostList posts={posts} />
      {hasFinished && (
        <p>You have reached the end!</p>
      )}
    </>
  )
}
```

4. Switch statements for multiple conditions

What if we are in a situation where have many different conditions, more than just one or two?

We could certainly write multiple if-statements, but all of these if statements, as we've seen earlier, are written above our returned JSX.

Too many if-statements can clutter our components. How do we make our code cleaner?

We can often extract multiple conditions to a separate component which contains a switch statement.

For example, we have a Menu component that we can toggle and display different tabs.

We have tabs that can display user, chat and room data as you see below:

```
export default function Menu() {
  const [menu, setMenu] = React.useState(1);

  function toggleMenu() {
    setMenu((m) => {
      if (m === 3) return 1;
      return m + 1;
    });
  }

  return (
    <>
      <MenuItem menu={menu} />
      <button onClick={toggleMenu}>Toggle Menu</button>
    </>
  );
}

function MenuItem({ menu }) {
  switch (menu) {
    case 1:
      return <Users />;
    case 2:
      return <Chats />;
    case 3:
      return <Rooms />;
    default:
      return null;
  }
}
```

Since we are using a dedicated MenuItem component with a switch statement, our parent Menu component is not cluttered by conditional logic and we can easily see what component will be displayed given the `menu` state.

5. Want conditionals as components? Try JSX Control Statements

It's greatly beneficial to be able to use plain JavaScript within our React components, but if you want even more declarative and straightforward conditionals, check out the React library JSX control statements.

You can bring it into your React projects by running the following command:

```
npm install --save-dev babel-plugin-jsx-control-statements
```

Additionally, you can list it in your .babelrc file like so:

```
{
  "plugins": ["jsx-control-statements"]
}
```

This is a Babel plugin that allows you to use React components directly within your JSX to write very easy to understand conditionals.

The best way to understand the use of such a library is by taking a look at an example. Let's rewrite one of our previous examples with the help of JSX control statements:

```
export default function App() {
  const { isLoading, isError, posts } = usePosts();

  return (
    <Choose>
      <When condition={isLoading}>
        <div>Loading...</div>
      </When>
      <When condition={isError}>
        <div>Error!</div>
      </When>
      <Otherwise>
        <PostList posts={posts} />
      </Otherwise>
    </Choose>
  );
}
```

You can see that there's no if or ternary statement in sight and we have a very readable component structure.

Give JSX control statements a try in your next React project and see if a library like this is for you.