

Title: React.js Commonly Faced Problems**Answer:**

1 – Not starting a component name with a capital letter

A React component must have a name which starts with a capital letter.

If the component name does not start with a capital letter, the component use will be treated as a *built-in* element such as a `div` or `span`.

For example:

```
class greeting extends React.Component {  
  // ...  
}
```

If you try to render `<greeting />`, React will ignore the above and you will get a warning:

Warning: The tag `<greeting>` is unrecognized in this browser. If you meant to render a React comp

The bigger problem here is when you decide to name your component `button` or `img`. React will ignore your component and just render a vanilla HTML `button` or `img` tag.



Note how the “My Awesome Button” was not rendered above and React just rendered an empty HTML button element. React will not warn you in this case.

2 – Using single quotes instead of back-ticks

Strings created with back-ticks (``...``) are different from strings created with single quotes (`'...'`).

On most keyboards, the back-tick (```) character can be typed using the key above the `tab` key.

We create a string using back-ticks when we need to include dynamic expressions inside that string (without resorting to string concatenation).

``This is a string template literal that can include expressions``

`'This is just a string, you cannot include expressions here'`

Let's say you want a string that always reports the current time:

"Time is ..."

```
// Current time string
const time = new Date().toLocaleTimeString();
```

```
// When using regular strings (single or double quotes),
// you need to use string concatenation:
'Time is ' + time
```

```
// When using back-ticks,
// you can inject the time in the string using ${}
`Time is ${time}`
```

Also, when using string literals (with back-ticks), you can create a string that spans multiple lines:

```
const template = `
```

```
CAN
```

```
SPAN
```

```
Multiple Lines`;
```

You can't do that with regular strings.

3— Using React.PropTypes

The `PropTypes` object was removed from React. It used to be available as `React.PropTypes` but you cannot use that anymore.

Instead, you need to:

1. Add the new **prop-types** package to your project: `npm install prop-types`
2. Import it: `import PropTypes from 'prop-types'`

Then you can use it. For example: `PropTypes.string`.

If you incorrectly use `React.PropTypes`, you'll get errors like:

```
TypeError: Cannot read property 'string' of undefined
```

4— Not using the right versions of what a tutorial is using

When watching or reading content about coding and following up with the examples they present, make sure that you are using correct versions of the tools the content is using. Usually, using the latest version of each tool is a safe bet, but if the content is a bit old you might run into some deprecation issues.

To be safe, stick with the major versions for the used tools. For example, if the tutorial is using React 16, don't follow up using React 15.

This is especially important for Node.js as well. You will face major problems if you use an older version of Node. For example, if you're following along with some tutorial that uses `Object.values` and you are using Node 6.x, that method did not exist back then. You need Node 7.x or higher.

5— Confusing functions with classes

Can you tell what's wrong with the following code?

```
class Numbers extends React.Component {  
  const arrayOfNumbers = _.range(1, 10);
```

```
  // ...  
}
```

The code above is invalid because inside the body of a JavaScript class you don't have the freedom of doing just about anything. You can only define methods and properties using limited syntax.

This is a bit confusing because the `{}` used in the class syntax looks like the plain-old block scope, but it's not.

Inside a function-based component, you DO have the freedom of doing just about anything:

```
// Totally Okay:
```

```
const Number = (props) => {  
  const arrayOfNumbers = _.range(1, 10);
```

```
  // ...  
};
```

6— Passing numbers as strings

You can pass a prop value with a string:

```
<Greeting name="World" />
```

If you need to pass a numeric value, don't use strings:

```
// Don't do this  
<Greeting counter="7" />
```

Instead, use curly braces to pass an actual numeric value:

```
// Do this instead  
<Greeting counter={7} />
```

Using `{7}`, inside the `Greeting` component, `this.props.counter` will have the actual numeric `7` value and it will be safe to do mathematical operations on that. If you pass it as `"7"` and then treat it as a number, you might run into unexpected results.

7—Forgetting that another app instance is still using the same port

To run a web server, you need to use a host (like `127.0.0.1`) and a port (like `8080`) to make the server listen for request on a valid http address.

Once the web server is running successfully, it has control over that port. You cannot use the same port for anything else. The port will be busy.

If you try to run the same server in another terminal, you'll get an error that the port is "in use". Something like:

```
Error: listen EADDRINUSE 127.0.0.1:8080
```

Be aware that sometimes a web server might be running in the *background* or inside a detached screen/tmux session. You don't see it, but it is still occupying the port. To restart your server, you need to "kill" the one that's still running.

To identify the process that's using a certain port, you can either use a command like `ps` (and `grep` for something about your app) or if you know the port number you can use the `lsof` command:

```
lsof -i :8080
```

8—Forgetting to create environment variables

Some projects depend on the existence of shell environment variables in order to start. If you run these projects without the needed environment variables, they will try to use undefined values for them and will potentially give you some cryptic errors.

For example, if a project connects to a database like MongoDB, chances are it uses an environment variable like `process.env.MONGO_URI` to connect to it. This allows the project to be used with different MongoDB instances in different environments.

To locally run the project that connects to a MongoDB, you have to export a `MONGO_URI` environment variable first. For example, if you have a local MongoDB running on port `27017`, you would need to do this before running the project:

```
export MONGO_URI="mongodb://localhost:27017/mydb"
```

You can *grep* the project source code for `process.env` to figure out what environment variables it needs in order to work correctly.

9— Confusing curly braces `{}` with parentheses `()`

Instead of:

```
return {  
  something();  
};
```

You need:

```
return (  
  something();  
);
```

The first one will try (and fail) to return an object while the second one will correctly call the `something()` function and return what that function returns.

Since any `<tag>` in JSX will translate to a function call, this problem applies when returning any JSX.

This problem is also common in arrow functions' *short* syntax.

Instead of:

```
const Greeting = () => {  
  <div>  
    Hello World  
  </div>  
};
```

You need:

```
const Greeting = () => (  
  <div>  
    Hello World  
  </div>  
);
```

When you use curly braces with an arrow function, you are starting the scope of that function. The short syntax of arrow functions does not use curly braces.

10 – Not wrapping objects with parentheses

The curly braces vs. parentheses problem above is also confusing when you want to create a short arrow function that returns a plain-old object.

Instead of:

```
const myAction = () => { type: 'DO_THIS' };
```

You need:

```
const myAction = () => ({ type: 'DO_THIS'});
```

Without wrapping the object in parentheses, you would not be using the short syntax. You will actually be defining a label for a string!

This is common inside the *updater* function of the `setState` method because it needs to return an object. You need to wrap that object with parenthesis if you want to use the short arrow function syntax.

Instead of:

```
this.setState(prevState => { answer: 42 });
```

You need:

```
this.setState(prevState => ({ answer: 42 }));
```

11 – Not using the right capitalization of API elements and props

It's `React.Component`, not `React.component`. It's `componentDidMount`, not `ComponentDidMount`. It's *usually* `ReactDOM`, not `ReactDom`.

Pay attention to the API capitalization that you need. If you use incorrect capitalization, the errors you'll get might not clearly state what the problem is.

When importing from `react` and `react-dom`, make sure that you're importing the correct names and what you're using is exactly the same as what you're importing. ESLint can help you point out what is not being used.

This problem is also common in accessing component props:

```
<Greeting userName="Max" />
```

```
// Inside the component, you need  
props.userName
```

If, instead of `props.userName`, you incorrectly use `props.username` or `props.UserName`, you will be using an undefined value. Pay attention to that, or better yet, make your ESLint configuration point these problems out as well.

12— Confusing the state object with instance properties

In a class component, you can define a local `state` object and later access it with `this`:

```
class Greeting extends React.Component {  
  state = {  
    name: "World",  
  };  
}
```

```
render() {  
  return `Hello ${this.state.name}`;  
}  
}
```

The above will output "Hello World".

You can also define other local instance properties beside `state`:

```
class Greeting extends React.Component {  
  user = {  
    name: "World",  
  };  
}
```

```
render() {  
  return `Hello ${this.user.name}`;  
}  
}
```

The above will also output "Hello World".

The `state` instance property is a special one because React will manage it. You can only change it through `setState` and React will *react* when you do. However, all other instance properties that you define will have no effect on the rendering algorithm. You can change `this.user` in the above example as you wish and React will not trigger a render cycle in React.

13— Confusing `<tag/>` with `</tag>`

Don't misplace the `/` character in your closing tags. Admittedly, sometimes you can use `<tag/>` and other times you need `</tag>`.

In HTML, there is something called a "self-closing tag" (AKA void tag). Those are the tags representing elements that do not have any children nodes. For example, the `img` tag is a self-closing one:

```

```

*// You don't use *

A `div` tag can have children, and so you use opening and closing tags:

```
<div>  
  Children here...  
</div>
```

The same applies to React components. If the component has children content, it might look like this:

```
<Greeting>Hello!</Greeting>
```

// Notice the position of the / character.

If the component does not have children, you can write it with open/closing tags or just a self-closing tag:

// 2 valid ways

```
<Greeting></Greeting>
```

```
<Greeting />
```

*// Notice how the / character moves based on whether the element
// is self-closing or not*

The following use is invalid:

// Wrong


```
<Greeting><Greeting />
```

If you misplace the `/` character, you will get errors like:

Syntax error: Unterminated JSX contents

14 – Assuming import/export will just work

The import/export feature is an official feature in JavaScript (since 2015). However, it's the only ES2015 feature that is not yet fully supported in modern browsers and the latest Node.

The popular configuration of a React project uses Webpack and Babel. Both allow the use of this feature and compile it down to something all browsers can understand. You can only use import/export if you have something like Webpack or Babel in your flow.

However, having import/export in your React bundled app does not mean that you can just use them anywhere you want! For example, if you're also doing server-side rendering through the latest Node, things will not work for you. You will most likely get an *"unexpected token"* error.

To have Node understand import/export as well (which is something you need if you use them on the front-end and you want to do SSR as well), you will have to run Node itself with a Babel preset (like the *env* preset) that can transpile them. You can use tools like *pm2*, *nodemon*, and *babel-watch* to do so in development and have Node restart every time you change something.

15 – Not binding handler methods

I saved this one for last because it's a big one and a very common problem.

You can define class methods in a React component and then use them in the component's `render` method. For example:

```
class Greeting extends React.Component {  
  whoIsThis() {  
    console.dir(this); // "this" is the caller of whoIsThis  
    return "World";  
  }  
}
```

```
render() {  
  return `Hello ${this.whoIsThis()}`;  
}  
}
```

```
ReactDOM.render(<Greeting />, mountNode);
```

I used the `whoIsThis` method inside the `render` method with `this.whoIsThis` because inside `render`, the `this` keyword refers to the component instance associated with the DOM element that represents the component.

React internally makes sure that “`this`” inside its class methods refers to the instance. However, JavaScript does not bind the instance automatically when you use a *reference* to the `whoIsThis` method.

The `console.dir` line in `whoIsThis` will correctly report the component instance because that method was called *directly* from the `render` method with an *explicit* caller (`this`). You should see the `Greeting` object in the console when you execute the code above:

The screenshot shows a code editor with the following code:

```

1 class Greeting extends React.Component {
2   whoIsThis() {
3     console.dir(this);
4     return "World";
5   }
6   render() {
7     return `Hello ${this.whoIsThis()}`
8   }
9 }
10
11 ReactDOM.render(<Greeting />, mountNode);
12

```

The browser window shows "Hello World". The console shows the `Greeting` object with the following structure:

```

Greeting {
  context: {},
  props: {},
  refs: {},
  state: null,
  updater: {isMounted: ...},
  _reactInternalFiber: ...,
  _reactInternalInstance: ...,
  isMounted: (...),
  replaceState: (...),
  __proto__: Component
}

```

However, when you use the same method in a *delayed-execution* channel, such as an *event handler*, the caller will no longer be explicit and the `console.dir` line will not report the component instance.

See the code and output (after clicking) below.

The screenshot shows a code editor with the following code:

```

1 class Greeting extends React.Component {
2   whoIsThis() {
3     console.dir(this);
4   }
5   render() {
6     return (
7       <div onClick={this.whoIsThis}>
8         Hello World
9       </div>
10    );
11   }
12 }
13
14 ReactDOM.render(<Greeting />, mountNode);
15

```

The browser window shows "Hello World". The console shows the message "Console was cleared" followed by "undefined".

In the code above, React invokes the `whoIsThis` method when you click on the string, but it will not give you access to the component instance inside of it. This is why you get `undefined` when we click the string. This is a problem if your class method needs access to things like `this.props` and `this.state`. It will simply not work.

There are many solutions for this problem. You can wrap the method in an inline function or use the `.bind` call to force the method to remember its caller. Both are okay for infrequently-updated components. You can also optimize the `bind` method by doing it in the *constructor* of the class instead of in the `render` method. However, the best solution to this method is to enable the ECMAScript class-fields feature (which is still stage-3) through Babel and just use an arrow function for the handlers:

```
class Greeting extends React.Component {  
  whoIsThis = () => {  
    console.dir(this);  
  }  
}
```

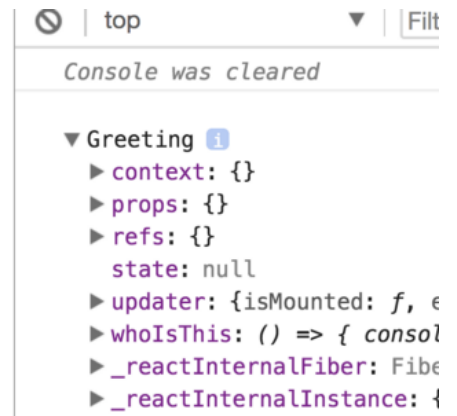
```
render() {  
  return (  
    <div onClick={this.whoIsThis}>  
      Hello World  
    </div>  
  );  
}
```

This will work as expected:



```
1 class Greeting extends React.Component {  
2   whoIsThis = () => {  
3     console.dir(this);  
4   }  
5   render() {  
6     return (  
7       <div onClick={this.whoIsThis}>  
8         Hello World  
9       </div>  
10    );  
11  }  
12 }  
13  
14 ReactDOM.render(<Greeting />, mountNode);  
15
```

Hello World



```
top | Filter  
Console was cleared  
▼ Greeting ⓘ  
  ► context: {}  
  ► props: {}  
  ► refs: {}  
  state: null  
  ► updater: {isMounted: f, ...}  
  ► whoIsThis: () => { console.dir(this); }  
  ► _reactInternalFiber: Fiber  
  ► _reactInternalInstance: {
```

That's all for now. Thanks for reading!

Tags: react