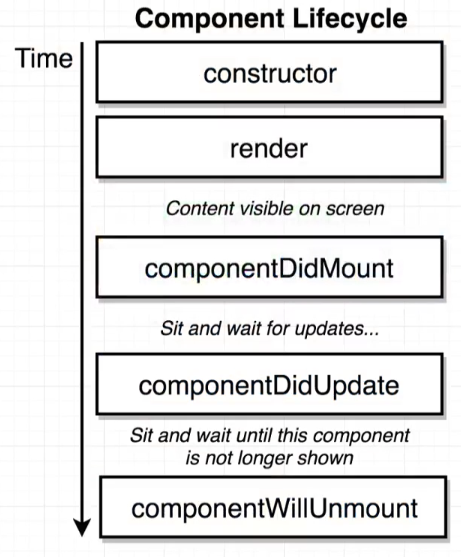
Understanding LIFECYCLE Methods –

We’ll be doing a little refactoring to our App component – second way on initializing state. In order to understand this other way of initializing state, we need to learn about lifecycle methods.



A component lifecycle method is a function that we can optionally define inside of our class-based components.

If we decide to implement these methods, they will be called automatically by React at certain points during a components lifecycle.

Lifecycle – entire series of events such as -- component created, shown on DOM, at some point we can call setState which causes re-rendering of component, at some point of time component may get removed from the DOM altogether.

**Render** method is one function that is NOT optional.

We start off with the constructor being called – then the render method gets called and returns some JSX that is visible on the screen.

After that, we see a series of different lifecycle methods being called at different points in time. **Immediately after a component shows up on the screen**, a lifecycle method called ‘componentDidMount()’ gets called.

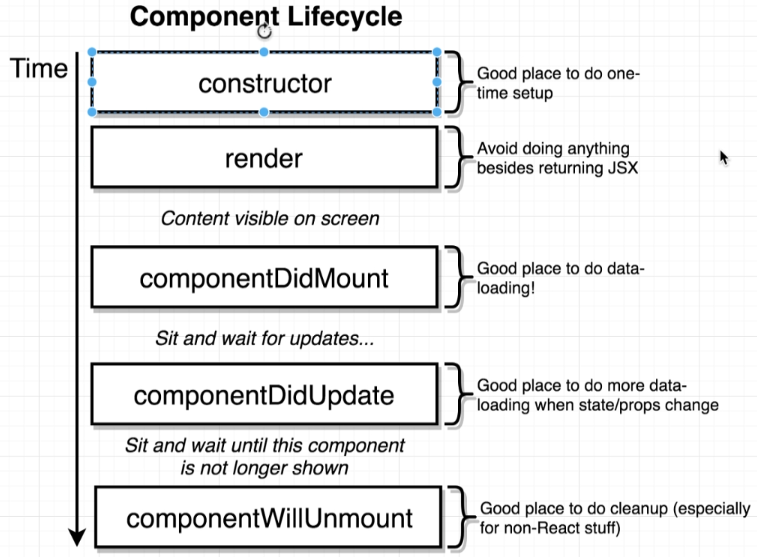
This means that if we define a function, outside the constructor and above the render method named componentDidMount, this function will automatically be called one time when our component first gets rendered onto the screen. So we can put some amount of code in here to set-up or do some initial data loading or some different operations that we might want to do one time when our component shows up.

After this method gets called, our component sits around and waits for an update – update is going to come in form of calling setState method. Anytime we update our state, component re-renders itself and lifecycle method ‘componentDidUpdate()’ gets automatically called.

At some point in time, we might want to stop showing this component on the screen – then ‘componentWillUnmount()’ methods gets automatically called. This method is usually used if we want to do some cleanup after our component. (to be discussed later)

Note – Anytime component did update gets called, right before it render method is called – renders JSX and then component did update gets called.

WHY TO USE LIFECYCLE METHODS?



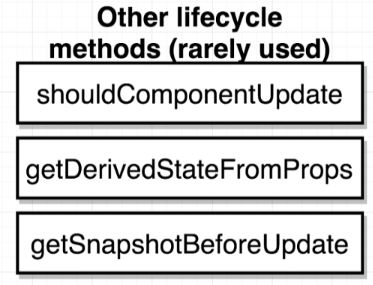
**Constructor** – Good place to do state initialization. Also some data loading can be done here (there’s a little caveat on that … told in did mount function)

**Render** – Returning some JSX only. (never make a network request or fetch user’s location)

**DidMount** – Perfect location to do some initial data loading for our component, or to start some outside process like getting the user’s location if we only have to do it one time. It is recommended that we do network related stuff here – if we centralize all data loading stuff inside component did mount method, as opposed to spreading it out in both constructor and did mount method, it’ll lead to a more clearer code. (technically we can do it in any of the two)

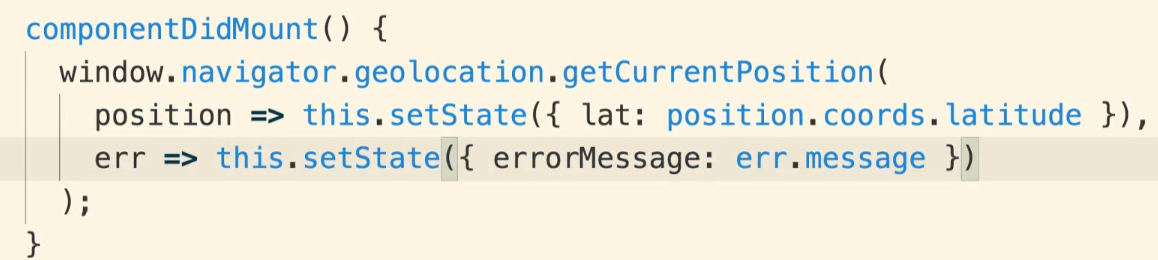
**DidUpdate** – Called every single time component is updated (state change or new props received from parent). Good location to do some data loading that needs to be done every single time a component gets updated.(new network request every time button is clicked or input comes)

**WillUnmount** – Remove component from screen and do a cleanup.



Very rarely used lifecycle methods.

Refactoring data loading code into componentDidMount function –



The **constructor**, now has only one purpose – initializing our state object. We can do a refactor here also to initialize state in a different manner. (we can only do this update as we’ve moved the data loading stuff in another function)

STATE-Initialization Refactor?

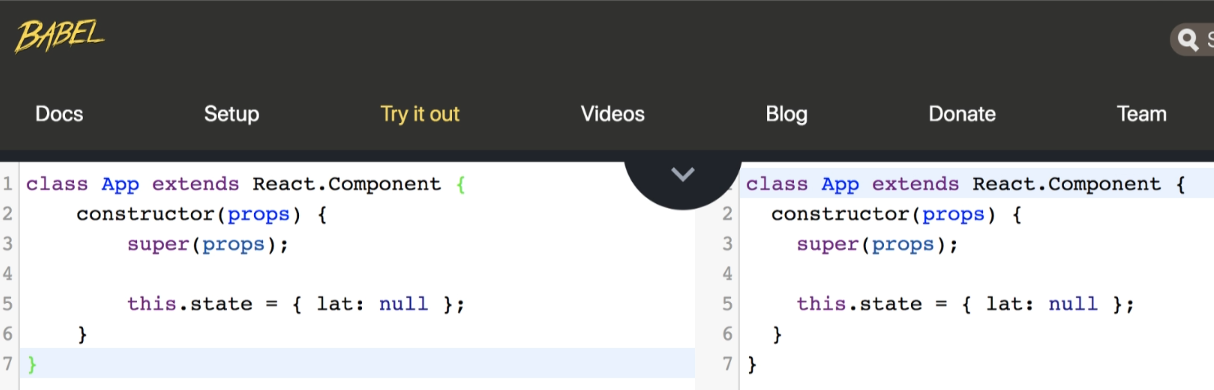


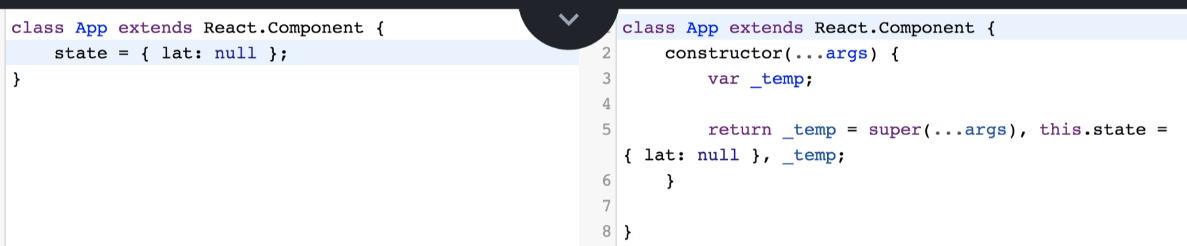
Outside the constructor function.

This single line is equivalent to defining the constructor function and initializing this.state inside of it.



Code written on our editor runs through tool called **Babel.**

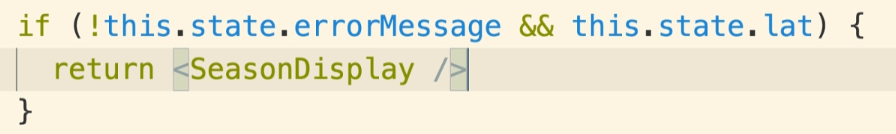




Babel implemented the constructor function for us automatically – and did exactly what we did using syntax 1(this.state = {initial values}).

PASSING State as PROPS?

Now, we’re going to import the season display component into the index.js file – and rather than printing the latitude, we want to show the season.

When returning the Season Display component, we must tell the component what latitude user is in.

The component will then figure out whether it’s summer or winter and show UI accordingly.

So, we have to take our latitude (i.e. inside of state object) and give value of it down to Season Display component – done using props system.

So, we’re taking a property from the state on the App component, and passing it as a prop down into season display.

Now, Season Display is going to be closely linked to the App component. Anytime we call setState inside of the parent component of the App, and update the latitude, the App component is going to re-render itself – that causes Season Display to be updated as well. (the lat props changes – new latitude value is going to be put into SD component & SD will be re-rendered as well)

Anytime we call setState, the component re-renders itself. In addition, component will also re-render any children it is showing as well.

Now we’ve got the latitude inside of Season Display component. Now, we’ll try to figure out Season based on latitude and month.

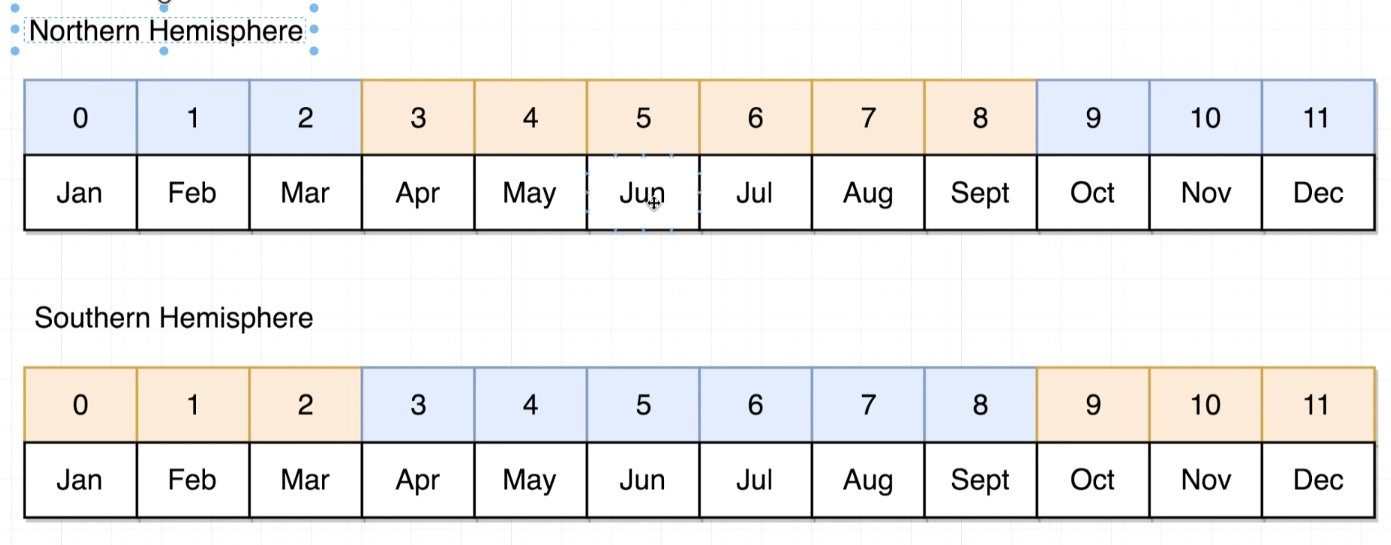
Determining Season?

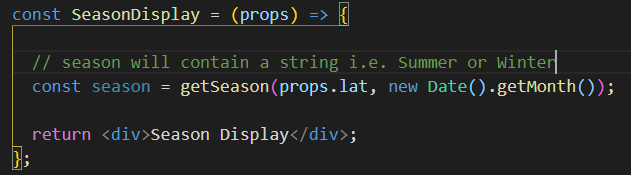
Using the latitude and month, we can decide whether it’s summer or winter.

To get the month, we use Date method in JavaScript.

[new Date().getMonth() – returns an integer indicating current month of the year]

* Date returned is a zero-indexed date. [Returns 1 => it’s the second month]





If we’re in Northern Hemisphere – implies that the latitude > 0.

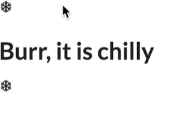
Now, if we’re b/w months 2 – 9, that means summer in Northern Hemisphere.

And vice versa logic for Southern Hemisphere.



Now, we just need to do styling inside of Season Display component.

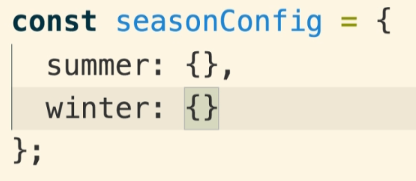




* We have duplicated ternary expression – can be refactored.
* We assigned name of the icon to a variable named icon – due to which we needed to make use of template string.

The refactor is going to be sort of a pattern to be followed always.

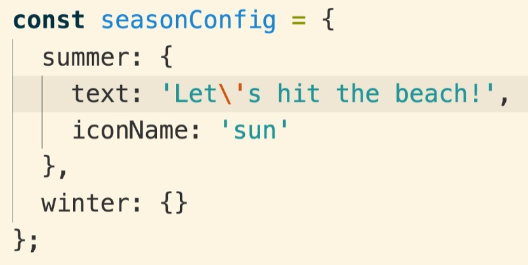
The IDEA is that we’re going to create a sort of **configuration object** at the top of the file –



At the top of the file, we’re going to create an object (called seasonConfig) – this is an object with two key-value pairs in it - one for each season.

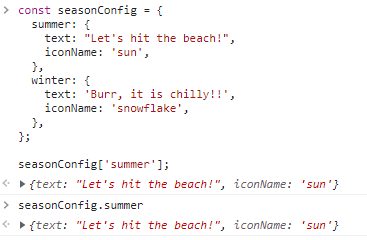
NOTE that the keys inside this object match up perfectly with the strings we’re returning from our getSeason function.

Now, these two objects are going to tell us what text and icon name we’re going to use in the specific season.



Now, to get at either of those seasonConfig objects – i.e. text and icon name for summer/winter, we can reference these inside our App component using –

seasonConfig[season] – returns an object {text,iconName} inside of it.



https://dmitripavlutin.com/access-object-properties-javascript/

We import the CSS file into our project –

When we import a CSS file into our JS file, it does not mean that we’re going to paste CSS into the JS file at that position.

BTS webpack – open source dependency inside our project that takes all these different files and joins them together – sees that we’re importing a CSS file. It is going to take the contents from there and stick it into the index.html file.

Root element inside the component has a class name equal to CSS class name of the component – pattern followed by many people.



Customizing Spinner (use of DEFAULT props)?

Anytime we want to customize how a component looks or behaves, we’re going to have to pass some props to it.



Now, if we forget to send a message, it’ll show an empty spinner rolling – we want to then show some default message like Loading.

Two ways to achieve this –

1. 
2. After we define the component, we can define something on the function itself by calling default props – Spinner.defaultProps = {object}

It’s going to provide some default properties to this component. If we forget to or decide not to put in a message when we create an instance of Spinner, we use the default props listed down here.



Let’s say in future, that for all cases (error, loading, season display), we want to have a red border around the page – we would need to wrap all the components with a div.



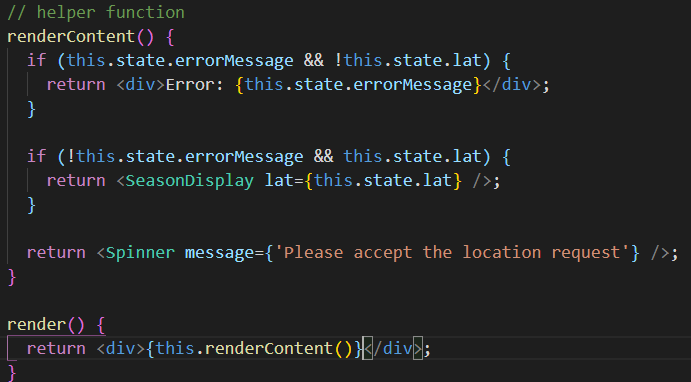
We want to work our way around this.

The root of our problems is that we have all this conditional logic inside of the render function.

So, we are going to use a helper function that contains all code currently inside render method.

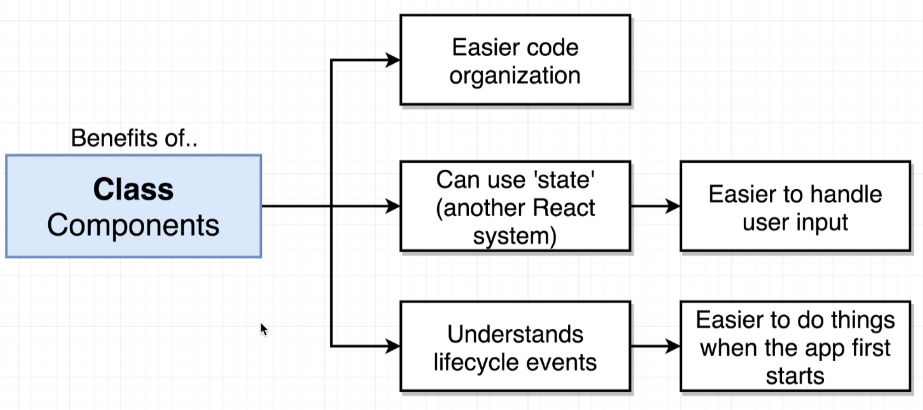
We still have the same logic as before but now it’s all being centralized in one location and changes in future can be easily made.

**Anytime we make a component, we try as much to not have multiple return statements inside render method.**



**If we ever have a conditional logic, we’re going to put it into a helper function always.**

RECAP –



Lecture 73 (0:00 – 2:00)