# What's a Lifecycle Method?

Lifecycle methods are methods that get called at certain moments in a component’s life.

You can write a lifecycle method that gets called right before a component renders for the first time.

You can write a lifecycle method that gets called right after a component renders, every time except for the first time.

You can attach lifecycle methods to a lot of different moments in a component’s life. This has powerful implications!

In this unit, we will walk through each lifecycle method. Click Next to begin!

**Mounting Lifecycle Methods**

There are three categories of lifecycle methods: *mounting*, *updating*, and *unmounting*. This lesson is about the first category: *mounting* lifecycle methods.

A component “mounts” when it renders for the first time. This is when *mounting lifecycle methods* get called.

There are three *mounting lifecycle methods:*

* componentWillMount
* render
* componentDidMount

When a component *mounts*, it automatically calls these three methods, in order.

**componentWillMount**

The first *mounting* lifecycle method is called componentWillMount.

When a component renders for the first time, componentWillMount gets called right *before* render.

Look at **Example.js**, and follow these steps:

1. On lines 14-17, <Example /> is rendered for the first time. <Example />‘s *mounting* period begins.
2. <Example /> calls the first mounting lifecycle method, componentWillMount.
3. componentWillMount executes, and an alert appears on the screen. (lines 5-7)
4. After componentWillMount has finished, <Example /> calls the second mounting lifecycle method: render.
5. <h1>Hello world</h1> appears on the screen (lines 9-11)
6. Two seconds later, <Example /> renders again (lines 20-22). componentWillMount does NOT get called, because mounting lifecycle events only execute the first time that a component renders.

You can call this.setState from within componentWillMount!

Look at **Example2.js** for an example of this.setState inside of componentWillMount. See if you can follow how <Example2 /> would render <h1>Hello world</h1>.

**Instructions**

**1.**

Select **Flashy.js**.

Take a look at the Flashy component class. Click Run.

You should see two alerts:

1. “AND NOW, FOR THE FIRST TIME EVER… FLASHY!!!!”.
2. “Flashy is rendering!”

Oooh, la la!

Checkpoint 2 Passed

**2.**

At the bottom of the file, delete the /\* and \*/ and uncomment the commented-out code.

This time, after <Flashy /> renders, two seconds will pass and then <Flashy /> will render again in a different color.

Checkpoint 3 Passed

**3.**

How utterly embarrassing!

<Flashy /> alerted, “AND NOW, FOR THE FIRST TIME EVER… “ *both times!*

The second rendering was not “the first time ever.” Everyone could tell that the second alert was a lie. The show was a bust, and Flashy received scathing reviews.

<Flashy /> needs to alert, “AND NOW, FOR THE FIRST TIME EVER…” ***only the first time*** that <Flashy /> renders.

If you need to do something *only the first time* that a component renders, then it’s probably a job for a mounting lifecycle method!

Before the render function, give Flashy a new property named componentWillMount. Set componentWillMount‘s *value* equal to a function:

componentWillMount() {}

render() { ...

Now, highlight the alert that should only happen once:

alert('AND NOW, FOR THE FIRST TIME EVER... FLASHY!!!!');

Cut this alert out of the render function, and paste it into the componentWillMount function instead.

Upon clicking Run, you should see “Flashy is rendering!” both times, but “AND NOW, FOR THE FIRST TIME EVER… FLASHY!!!!” *only the first time.* Phew!

# render

render is a lifecycle method!

We won’t go over render here - we’ve already talked about it plenty. However, you should understand how render fits into the mounting period. Whenever a component mounts, componentWillMount is called first, followed by render, followed by componentDidMount.

render belongs to two categories: mounting lifecycle methods, and updating lifecycle methods. We’ll cover updating lifecycle methods in the next lesson. But first, there’s one final mounting lifecycle method!

**componentDidMount**

The final mounting lifecycle method is called componentDidMount.

When a component renders for the first time, componentDidMount gets called right *after* the HTML from render has finished loading. Look in the code editor for an example of componentDidMount.

componentDidMount gets used a lot!

If your React app uses AJAX to fetch initial data from an API, then componentDidMount is the place to make that AJAX call. More generally, componentDidMount is a good place to connect a React app to external applications, such as web APIs or JavaScript frameworks. componentDidMount is also the place to set timers using setTimeout or setInterval.

If that sounds vague, don’t worry. You’ll put lifecycle methods into practice in this course’s final project! Not to mention in the real world…

**Instructions**

**1.**

Take a look at **Flashy.js**.

Flashy is mostly the same component class from earlier, but it has a brand new alert added to its render function on line 13.

Click Run and watch Flashy flash. You should see three alerts:

1. “AND NOW, FOR THE FIRST TIME EVER… FLASHY!!!!”
2. “Flashy is rendering!”
3. “YOU JUST WITNESSED THE DEBUT OF… FLASHY!!!!!!!”

Thanks to setTimeout, <Flashy /> will render twice, with a two-second pause in between.

Checkpoint 2 Passed

**2.**

How incredibly gauche!

<Flashy /> alerted, “YOU JUST WITNESSED THE DEBUT OF… FLASHY!!!!!!!” *both times!* The second time was *not* a debut, and many guests stormed out in a great huff.

You need to alert “YOU JUST WITNESSED THE DEBUT OF… FLASHY!!!!!!!” *only the first time* that <Flashy /> renders.

You can’t use componentWillMount for this, because you want your alert to happen right *after* <Flashy /> renders, not before. It would be weird to say, “you just witnessed the debut!” before it happens.

You want something to happen right *after* the very first rendering… that’s exactly what componentDidMount is for!

In between componentWillMount and render, give Flashy an empty componentDidMount method:

componentDidMount() {}

render() { ...

Checkpoint 3 Passed

**3.**

Inside of the render function, highlight the offending alert:

alert('YOU JUST WITNESSED THE DEBUT OF... FLASHY!!!!!!!');

Cut this alert out of the render function, and paste it into the componentDidMount function instead.

Now when you hit Run, you should see the alert “Flashy is rendering!” both times, but the other alerts should appear in the first time only. Crisis averted!

# Updating Lifecycle Methods

There are two categories that we haven’t yet discussed: updating and unmounting lifecycle methods. This lesson covers both.

What is updating?

The first time that a component instance renders, it does not update. A component updates every time that it renders, starting with the second render.

There are five updating lifecycle methods:

* componentWillReceiveProps
* shouldComponentUpdate
* componentWillUpdate
* render
* componentDidUpdate

Whenever a component instance updates, it automatically calls all five of these methods, in order.

# componentWillReceiveProps

The first updating lifecycle method is called componentWillReceiveProps.

When a component instance updates, componentWillReceiveProps gets called before the rendering begins.

As one might expect, componentWillReceiveProps only gets called if the component will receive props:

// componentWillReceiveProps will get called here:

ReactDOM.render(

<Example prop="myVal" />,

document.getElementById('app')

);

// componentWillReceiveProps will NOT get called here:

ReactDOM.render(

<Example />,

document.getElementById('app')

);

Look in the code editor for an example of componentWillReceiveProps. Read it through and try to figure out how it works.

componentWillReceiveProps automatically gets passed one argument: an object called nextProps. nextProps is a preview of the upcoming props object that the component is about to receive.

On line 6, nextProps.text will evaluate to "Hello world".

**Instructions**

**1.**

Select **App.js**. Click Run. You’ll see a game load in the browser.

The goal of this game is to click on high numbers. Every click has to be a higher number than the previous click.

The game has some problems. The numbers move around in a jerky way. “Top Number” is broken. Worst of all, the game never ends!

We’ll use lifecycle methods to fix these problems.

Start a new game. When you’re done, click the refresh button in the top left of the browser column to make the game stop.

Checkpoint 2 Passed

**2.**

“Top Number” is supposed to show the highest number that you’ve clicked so far.

It isn’t working! No matter what number you click, “Top Number” thinks that that number is the highest number so far.

Select **TopNumber.js**. Take a look at the TopNumber component class.

In between constructor and render, give TopNumber a new method named componentWillReceiveProps.

Give componentWillReceiveProps a parameter of nextProps.

Checkpoint 3 Passed

**3.**

Look at TopNumber‘s render function. The problem lies in between the <h1></h1> tags.

Whenever you click on a number in the browser, that number gets passed from App, to TopNumber, as this.props.number. In between the <h1></h1> tags, you can see that TopNumber displays every this.props.number that it gets.

When TopNumber gets a new this.props.number, you don’t always want it displayed! You want to check whether that number is, in fact, the highest number yet. If it is the highest, only then should it get displayed in the <h1></h1>.

componentWillReceiveProps can help!

Whenever you get a new this.props.number, componentWillReceiveProps sees it before render does. componentWillReceiveProps can scan this new this.props.number, and decide whether it should be displayed.

Here’s a starting point:

componentWillReceiveProps: function (nextProps) {

if (nextProps.number > this.state.highest) {

// nextProps.number is the new highest number so far!

// Display it!

}

},

Checkpoint 4 Passed

**4.**

Look at at the constructor. this.state.highest starts equal to 0… and never changes! It will always be 0.

You need this.state.highest to live up to its name and keep track of the highest number so far.

In componentWillReceiveProps, remove these lines:

// nextProps.number is the new highest number so far!

// Display it!

In their place, update this.state.highest:

this.setState({

highest: nextProps.number

});

Checkpoint 5 Passed

**5.**

Good! Now whenever TopNumber gets a new this.props.number, componentWillReceiveProps will use it to keep track of the highest number so far.

All that remains is to display the results!

In TopNumber‘s render function, replace {this.props.number} with {this.state.highest}.

Checkpoint 6 Passed

**6.**

Try playing another round. It’s still definitely broken, but Top Number should work now!

This is a common use of componentWillReceiveProps: comparing incoming props to current props or state, and deciding what to render based on that comparison.

If you are already comfortable with React, then you may have caught a detail about this example that is considered bad form. this.state.highest is [derived from props](https://facebook.github.io/react/tips/props-in-getInitialState-as-anti-pattern.html). That means that we use information from props to set the value of information stored in state. We’ll go into this more in the next course!

# shouldComponentUpdate

The second updating lifecycle method is called shouldComponentUpdate.

When a component updates, shouldComponentUpdate gets called after componentWillReceiveProps, but still before the rendering begins.

Look at **Example.js** in the code editor. Read it through and try to figure out how shouldComponentUpdate works.

shouldComponentUpdate should return either true or false.

If shouldComponentUpdate returns true, then nothing noticeable happens. But if shouldComponentUpdate returns false, then the component will not update! None of the remaining lifecycle methods for that updating period will be called, including render.

The best way to use shouldComponentUpdate is to have it return false only under certain conditions. If those conditions are met, then your component will not update.

shouldComponentUpdate automatically receives two arguments: nextProps and nextState. It’s typical to compare nextProps and nextState to the current this.props and this.state, and use the results to decide what to do. See how Example.js does this on lines 10-19.

**Instructions**

**1.**

Select **Target.js** and look at the Target component class.

A <Target /> renders a number in a random location. In this game, a “target” is a number that you try to click on.

Whenever a new target appears, they all rerender. This is a problem!

When a target renders, it picks a random location. When a target rerenders, it picks a new random location! This is why the numbers move in such a jerky way: whenever a new one appears, all of the old ones move to a new random place.

When a new target appears, you want all of the old targets to stay where they are! This will make the gameplay much nicer.

You can use shouldComponentUpdate to make Target only rerender when it actually needs to.

Give Target a new method named shouldComponentUpdate which takes two parameters: nextProps and nextState.

Checkpoint 2 Passed

**2.**

You want shouldComponentUpdate to return true when a target first appears. This will cause Target to update as usual.

You want shouldComponentUpdate to return false when a target has already rendered, and is about to repeat the same number as its last render. This will cause Target to cancel its update.

Inside the body of the shouldComponentUpdate function, write the following code:

return this.props.number != nextProps.number;

Click Run and try a new game. It should be much smoother!

# componentWillUpdate

The third updating lifecycle method is componentWillUpdate.

componentWillUpdate gets called in between shouldComponentUpdate and render.

componentWillUpdate receives two arguments: nextProps and nextState. Read Example in the code editor to see it in action.

You cannot call this.setState from the body of componentWillUpdate! Which begs the question, why would you use it?

The main purpose of componentWillUpdate is to interact with things outside of the React architecture. If you need to do non-React setup before a component renders, such as checking the window size or interacting with an API, then componentWillUpdate is a good place to do that.

If that sounds abstract, that’s okay! All of the lifecycle methods might feel a bit theoretical, until you’ve used them in real-life scenarios. You’ll be doing more of that in the next course.

**Instructions**

**1.**

Let’s use componentWillUpdate to make the screen’s background turn yellow if your top number breaks 950,000.

You will use this expression to change the screen’s background:

document.body.style.background = (new color goes here);

componentWillUpdate is a good place for setup code like this, that engages with the non-React world. document.body has nothing to do with React at all.

Open **TopNumber.js**.

Give TopNumber a new method named componentWillUpdate which takes two parameters: nextProps and nextState.

Checkpoint 2 Passed

**2.**

On every render, you want componentWillUpdate to check whether the background is yellow already. If it isn’t, then you want to check whether the top number is at least 950,000. If it is, then make the background yellow.

Add the following code to the body of componentWillUpdate:

if (document.body.style.background != yellow

&& this.state.highest >= 950\*1000) {

document.body.style.background = yellow;

}

Make sure not to put yellow in quotes! You are referencing the variable defined on line 3.

Of course, you’ll also need to change the color back to white when a user starts a new game! You can do this with the help of this.props.game, a boolean that reports whether or not a current game is in progress.

Expand your if statement into an else/if:

if (document.body.style.background != yellow

&& this.state.highest >= 950\*1000) {

document.body.style.background = yellow;

} else if (!this.props.game

&& nextProps.game) {

document.body.style.background = 'white';

}

Click Run and the let the browser refresh. See if you can get the yellow background.

# componentDidUpdate

The last updating lifecycle method is componentDidUpdate.

When a component instance updates, componentDidUpdate gets called after any rendered HTML has finished loading.

Look at Example for an example of componentDidUpdate.

componentDidUpdate automatically gets passed two arguments: prevProps and prevState. prevProps and prevState are references to the component’s props and state before the current updating period began. You can compare them to the current props and state.

componentDidUpdate is usually used for interacting with things outside of the React environment, like the browser or APIs. It’s similar to componentWillUpdate in that way, except that it gets called after render instead of before.

**Instructions**

**1.**

In the code editor, select **App.js**.

We can finally add the most important fix: making the game end!

You can see on lines 60-70 that there is already an endGame method set up that will cause the game to stop. You just have to decide when to call it.

You want the game to end when a user clicks a number less than their previous click. componentDidUpdate can do that!

Look at App’s constructor. You can see that App has a this.state.latestClick property. this.state.latestClick stores the value of the most recently clicked number.

This is the value that you need! You’re trying to compare the most recent click with the second-to-most-recent click.

Give App a method called componentDidUpdate which takes two parameters: prevProps and prevState.

Set body of this method to the following:

if (this.state.latestClick < prevState.latestClick) {

this.endGame();

}

Click Run and play a new game. So much better!

# componentWillUnmount

A component’s unmounting period occurs when the component is removed from the DOM. This could happen if the DOM is rerendered without the component, or if the user navigates to a different website or closes their web browser.

componentWillUnmount is the only unmounting lifecycle method!

componentWillUnmount gets called right before a component is removed from the DOM. If a component initiates any methods that require cleanup, then componentWillUnmount is where you should put that cleanup.

You can see an example in **Example.js**, as usual.

**Instructions**

**1.**

You are so sick of having to type exclamation points! You have to hold down the shift key, and then reach up with your pinkie… ugh. The worst.

You’ve written a program to type your exclamation points for you. Select **Enthused.js** and take a look at your source code.

On lines 4-8, Enthused contains a componentDidMount method that calls a setInterval function, which adds an exclamation point to some text every 15ms.

Enthused renders a ‘Stop!’ button. Clicking ‘Stop!’ will unmount the Enthused component and stop the exclamation points.

Open **App.js**. App renders an <h1></h1>, a text entry field, and a button. At first, this button says ‘Add Enthusiasm!’ Clicking on this button will remove it and replace it with an <Enthused /> instance. Clicking on <Enthused />‘s “Stop!” button will remove <Enthused /> and replace it with the first button.

Click Run. Let the browser refresh.

If this is your first time through this checkpoint, then try entering some text. Click on the button for auto-enthusiasm.

Checkpoint 2 Passed

**2.**

AAAAAH! The Stop button doesn’t work! Too much enthusiasm forever!!!!

Hit cmd-R to refresh the page.

Checkpoint 3 Passed

**3.**

Select **Enthused.js**.

Give Enthused a new componentWillUnmount method, that will stop the madness whenever Enthused is unmounted:

componentWillUnmount(prevProps, prevState) {

clearInterval(this.interval);

}

Checkpoint 4 Passed

**4.**

Try entering some text, and then adding some more enthusiasm. This time the Stop button should work!

# Lifecycle Methods Recap

Congratulations on making it to the end of our Introductory React courses! By this point, you have acquired all of the tools that you need to program in React!

However, you aren’t quite done with this course yet! You might be asking yourself how to create a React app on your own computer. We’ve created an article to walk you through this process. [Click here](https://www.codecademy.com/articles/how-to-create-a-react-app) to get started!