Bootstrap

1. Use Responsive Design with Bootstrap Fluid Containers

Description

In the HTML5 and CSS section of freeCodeCamp we built a Cat Photo App. Now let's go back to it. This time, we'll style it using the popular Bootstrap responsive CSS framework. Bootstrap will figure out how wide your screen is and respond by resizing your HTML elements - hence the name `Responsive Design` . With responsive design, there is no need to design a mobile version of your website. It will look good on devices with screens of any width. You can add Bootstrap to any app by adding the following code to the top of your HTML: `<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous"/>` In this case, we've already added it for you to this page behind the scenes. Note that using either `>` or `/>` to close the `link` tag is acceptable. To get started, we should nest all of our HTML (except the `link` tag and the `style` element) in a `div` element with the class `container-fluid` .

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<h2 class="red-text">CatPhotoApp</h2>

<p>Click here for <a href="#">cat photos</a>.</p>

<a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

<p>Things cats love:</p>
<ul>
  <li>cat nip</li>
  <li>laser pointers</li>
  <li>lasagna</li>
</ul>
```

```
<p>Top 3 things cats hate:</p>
<ol>
  <li>flea treatment</li>
  <li>thunder</li>
  <li>other cats</li>
</ol>
<form action="/submit-cat-photo">
  <label><input type="radio" name="indoor-outdoor"> Indoor</label>
  <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
  <label><input type="checkbox" name="personality"> Loving</label>
  <label><input type="checkbox" name="personality"> Lazy</label>
  <label><input type="checkbox" name="personality"> Crazy</label>
  <input type="text" placeholder="cat photo URL" required>
  <button type="submit">Submit</button>
</form>
```

Solution

```
// solution required
```

2. Make Images Mobile Responsive

Description

First, add a new image below the existing one. Set its `src` attribute to `https://bit.ly/fcc-running-cats` . It would be great if this image could be exactly the width of our phone's screen. Fortunately, with Bootstrap, all we need to do is add the `img-responsive` class to your image. Do this, and the image should perfectly fit the width of your page.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
```

```
cute orange cat lying on its back."></a>

  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

3. Center Text with Bootstrap

Description

Now that we're using Bootstrap, we can center our heading element to make it look better. All we need to do is add the class `text-center` to our `h2` element. Remember that you can add several classes to the same element by separating each of them with a space, like this: `<h2 class="red-text text-center">your text</h2>`

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
```

```
the camera.">
  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

4. Create a Bootstrap Button

Description

Bootstrap has its own styles for `button` elements, which look much better than the plain HTML ones. Create a new `button` element below your large kitten photo. Give it the `btn` and `btn-default` classes, as well as the text of "Like".

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
```

```
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">

  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
<html>
<head>
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>
</head>
<body>
<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">

   <!-- ADD Bootstrap Styled Button -->
  <button class="btn btn-default">Like</button>
```

```
    <p>Things cats love:</p>
    <ul>
      <li>cat nip</li>
      <li>laser pointers</li>
      <li>lasagna</li>
    </ul>
    <p>Top 3 things cats hate:</p>
    <ol>
      <li>flea treatment</li>
      <li>thunder</li>
      <li>other cats</li>
    </ol>
    <form action="/submit-cat-photo">
      <label><input type="radio" name="indoor-outdoor"> Indoor</label>
      <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
      <label><input type="checkbox" name="personality"> Loving</label>
      <label><input type="checkbox" name="personality"> Lazy</label>
      <label><input type="checkbox" name="personality"> Crazy</label>
      <input type="text" placeholder="cat photo URL" required>
      <button type="submit">Submit</button>
    </form>
  </div>
  </html>
```

5. Create a Block Element Bootstrap Button

Description

Normally, your `button` elements with the `btn` and `btn-default` classes are only as wide as the text that they contain. For example: `<button class="btn btn-default">Submit</button>` This button would only be as wide as the word "Submit". Submit By making them block elements with the additional class of `btn-block` , your button will stretch to fill your page's entire horizontal space and any elements following it will flow onto a "new line" below the block. `<button class="btn btn-default btn-block">Submit</button>` This button would take up 100% of the available width. Submit Note that these buttons still need the `btn` class. Add Bootstrap's `btn-block` class to your Bootstrap button.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>
```

```
<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <button class="btn btn-default">Like</button>
  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

6. Taste the Bootstrap Button Color Rainbow

Description

The `btn-primary` class is the main color you'll use in your app. It is useful for highlighting actions you want your user to take. Replace Bootstrap's `btn-default` class by `btn-primary` in your button. Note that this button will still need the `btn` and `btn-block` classes.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }
  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <button class="btn btn-default btn-block">Like</button>
  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

7. Call out Optional Actions with btn-info

Description

Bootstrap comes with several pre-defined colors for buttons. The `btn-info` class is used to call attention to optional actions that the user can take. Create a new block-level Bootstrap button below your "Like" button with the text "Info", and add Bootstrap's `btn-info` and `btn-block` classes to it. Note that these buttons still need the `btn` and `btn-block` classes.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <button class="btn btn-block btn-primary">Like</button>
  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

8. Warn Your Users of a Dangerous Action with btn-danger

Description

Bootstrap comes with several pre-defined colors for buttons. The `btn-danger` class is the button color you'll use to notify users that the button performs a destructive action, such as deleting a cat photo. Create a button with the text "Delete" and give it the class `btn-danger` . Note that these buttons still need the `btn` and `btn-block` classes.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <button class="btn btn-block btn-primary">Like</button>
  <button class="btn btn-block btn-info">Info</button>
  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

9. Use the Bootstrap Grid to Put Elements Side By Side

Description

Bootstrap uses a responsive 12-column grid system, which makes it easy to put elements into rows and specify each element's relative width. Most of Bootstrap's classes can be applied to a `div` element. Bootstrap has different column width attributes that it uses depending on how wide the user's screen is. For example, phones have narrow screens, and laptops have wider screens. Take for example Bootstrap's `col-md-*` class. Here, `md` means medium, and `*` is a number specifying how many columns wide the element should be. In this case, the column width of an element on a medium-sized screen, such as a laptop, is being specified. In the Cat Photo App that we're building, we'll use `col-xs-*`, where `xs` means extra small (like an extra-small mobile phone screen), and `*` is the number of columns specifying how many columns wide the element should be. Put the `Like`, `Info` and `Delete` buttons side-by-side by nesting all three of them within one `<div class="row">` element, then each of them within a `<div class="col-xs-4">` element. The `row` class is applied to a `div`, and the buttons themselves can be nested within it.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards the camera.">
  <button class="btn btn-block btn-primary">Like</button>
  <button class="btn btn-block btn-info">Info</button>
  <button class="btn btn-block btn-danger">Delete</button>
  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>

  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary">Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info">Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger">Delete</button>
    </div>
  </div>

  <p>Things cats love:</p>
```

```
    <ul>
      <li>cat nip</li>
      <li>laser pointers</li>
      <li>lasagna</li>
    </ul>
    <p>Top 3 things cats hate:</p>
    <ol>
      <li>flea treatment</li>
      <li>thunder</li>
      <li>other cats</li>
    </ol>
    <form action="/submit-cat-photo">
      <label><input type="radio" name="indoor-outdoor"> Indoor</label>
      <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
      <label><input type="checkbox" name="personality"> Loving</label>
      <label><input type="checkbox" name="personality"> Lazy</label>
      <label><input type="checkbox" name="personality"> Crazy</label>
      <input type="text" placeholder="cat photo URL" required>
      <button type="submit">Submit</button>
    </form>
  </div>
```

10. Ditch Custom CSS for Bootstrap

Description

We can clean up our code and make our Cat Photo App look more conventional by using Bootstrap's built-in styles instead of the custom styles we created earlier. Don't worry - there will be plenty of time to customize our CSS later. Delete the `.red-text` , `p` , and `.smaller-image` CSS declarations from your `style` element so that the only declarations left in your `style` element are `h2` and `thick-green-border` . Then delete the `p` element that contains a dead link. Then remove the `red-text` class from your `h2` element and replace it with the `text-primary` Bootstrap class. Finally, remove the "smaller-image" class from your first `img` element and replace it with the `img-responsive` class.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  .red-text {
    color: red;
  }

  h2 {
    font-family: Lobster, Monospace;
  }

  p {
    font-size: 16px;
    font-family: Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

  .smaller-image {
    width: 100px;
  }
</style>

<div class="container-fluid">
  <h2 class="red-text text-center">CatPhotoApp</h2>
  <p>Click here for <a href="#">cat photos</a>.</p>

  <a href="#"><img class="smaller-image thick-green-border" src="https://bit.ly/fcc-relaxing-cat" alt="A
cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary">Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info">Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger">Delete</button>
    </div>
  </div>
  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

11. Use a span to Target Inline Elements

Description

You can use spans to create inline elements. Remember when we used the `btn-block` class to make the button fill the entire row? normal button btn-block button That illustrates the difference between an "inline" element and a "block" element. By using the inline `span` element, you can put several elements on the same line, and even style different parts of the same line differently. Nest the word "love" in your "Things cats love" element below within a `span` element. Then give that `span` the class `text-danger` to make the text red. Here's how you would do this with the "Top 3 things cats hate" element: `<p>Top 3 things cats <span class="text-danger">hate:</span></p>`

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
```

```
  h2 {
    font-family: Lobster, Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

</style>

<div class="container-fluid">
  <h2 class="text-primary text-center">CatPhotoApp</h2>

  <a href="#"><img class="img-responsive thick-green-border" src="https://bit.ly/fcc-relaxing-cat"
alt="A cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary">Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info">Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger">Delete</button>
    </div>
  </div>
  <p>Things cats love:</p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

12. Create a Custom Heading

Description

We will make a simple heading for our Cat Photo App by putting the title and relaxing cat image in the same row. Remember, Bootstrap uses a responsive grid system, which makes it easy to put elements into rows and specify each element's relative width. Most of Bootstrap's classes can be applied to a `div` element. Nest your first image and your `h2` element within a single `<div class="row">` element. Nest your `h2` element within a `<div class="col-xs-8">`

and your image in a `<div class="col-xs-4">` so that they are on the same line. Notice how the image is now just the right size to fit along the text?

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">

<style>
  h2 {
    font-family: Lobster, Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }
</style>

<div class="container-fluid">
  <h2 class="text-primary text-center">CatPhotoApp</h2>

  <a href="#"><img class="img-responsive thick-green-border" src="https://bit.ly/fcc-relaxing-cat"
alt="A cute orange cat lying on its back."></a>

  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary">Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info">Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger">Delete</button>
    </div>
  </div>
  <p>Things cats <span class="text-danger">love:</span></p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

13. Add Font Awesome Icons to our Buttons

Description

Font Awesome is a convenient library of icons. These icons are vector graphics, stored in the `.svg` file format. These icons are treated just like fonts. You can specify their size using pixels, and they will assume the font size of their parent HTML elements. You can include Font Awesome in any app by adding the following code to the top of your HTML: `<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-awesome/4.5.0/css/font-awesome.min.css" integrity="sha384-XdYbMnZ/QjLh6iI4ogqCTaIjrFk87ip+ekIjefZch0Y+PvJ8CDYtEs1ipDmPorQ+" crossorigin="anonymous">` In this case, we've already added it for you to this page behind the scenes. The `i` element was originally used to make other elements italic, but is now commonly used for icons. You can add the Font Awesome classes to the `i` element to turn it into an icon, for example: `<i class="fa fa-info-circle"></i>` Note that the `span` element is also acceptable for use with icons. Use Font Awesome to add a `thumbs-up` icon to your like button by giving it an `i` element with the classes `fa` and `fa-thumbs-up`; make sure to keep the text "Like" next to the icon.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  h2 {
    font-family: Lobster, Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }
</style>

<div class="container-fluid">
  <div class="row">
    <div class="col-xs-8">
      <h2 class="text-primary text-center">CatPhotoApp</h2>
    </div>
    <div class="col-xs-4">
      <a href="#"><img class="img-responsive thick-green-border" src="https://bit.ly/fcc-relaxing-cat"
alt="A cute orange cat lying on its back."></a>
    </div>
  </div>
  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary">Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info">Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger">Delete</button>
    </div>
  </div>
  <p>Things cats <span class="text-danger">love:</span></p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

14. Add Font Awesome Icons to all of our Buttons

Description

Font Awesome is a convenient library of icons. These icons are vector graphics, stored in the `.svg` file format. These icons are treated just like fonts. You can specify their size using pixels, and they will assume the font size of their parent HTML elements. Use Font Awesome to add an `info-circle` icon to your info button and a `trash` icon to your delete button. Note: The `span` element is an acceptable alternative to the `i` element for the directions below.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  h2 {
    font-family: Lobster, Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }
</style>

<div class="container-fluid">
  <div class="row">
    <div class="col-xs-8">
      <h2 class="text-primary text-center">CatPhotoApp</h2>
    </div>
    <div class="col-xs-4">
      <a href="#"><img class="img-responsive thick-green-border" src="https://bit.ly/fcc-relaxing-cat"
alt="A cute orange cat lying on its back."></a>
    </div>
  </div>
  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary"><i class="fa fa-thumbs-up"></i> Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info">Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger">Delete</button>
    </div>
  </div>
  <p>Things cats <span class="text-danger">love:</span></p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

15. Responsively Style Radio Buttons

Description

You can use Bootstrap's `col-xs-*` classes on `form` elements, too! This way, our radio buttons will be evenly spread out across the page, regardless of how wide the screen resolution is. Nest both your radio buttons within a `<div class="row">` element. Then nest each of them within a `<div class="col-xs-6">` element. **Note:** As a reminder, radio buttons are `input` elements of type `radio` .

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  h2 {
    font-family: Lobster, Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }
</style>

<div class="container-fluid">
  <div class="row">
    <div class="col-xs-8">
      <h2 class="text-primary text-center">CatPhotoApp</h2>
    </div>
    <div class="col-xs-4">
      <a href="#"><img class="img-responsive thick-green-border" src="https://bit.ly/fcc-relaxing-cat"
alt="A cute orange cat lying on its back."></a>
    </div>
  </div>
  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary"><i class="fa fa-thumbs-up"></i> Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info"><i class="fa fa-info-circle"></i> Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger"><i class="fa fa-trash"></i> Delete</button>
    </div>
  </div>
  <p>Things cats <span class="text-danger">love:</span></p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <label><input type="radio" name="indoor-outdoor"> Indoor</label>
    <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

16. Responsively Style Checkboxes

Description

Since Bootstrap's `col-xs-*` classes are applicable to all `form` elements, you can use them on your checkboxes too! This way, the checkboxes will be evenly spread out across the page, regardless of how wide the screen resolution is.

Instructions

Nest all three of your checkboxes in a `<div class="row">` element. Then nest each of them in a `<div class="col-xs-4">` element.

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  h2 {
    font-family: Lobster, Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

</style>

<div class="container-fluid">
  <div class="row">
    <div class="col-xs-8">
      <h2 class="text-primary text-center">CatPhotoApp</h2>
    </div>
    <div class="col-xs-4">
      <a href="#"><img class="img-responsive thick-green-border" src="https://bit.ly/fcc-relaxing-cat"
alt="A cute orange cat lying on its back."></a>
    </div>
  </div>
  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary"><i class="fa fa-thumbs-up"></i> Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info"><i class="fa fa-info-circle"></i> Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger"><i class="fa fa-trash"></i> Delete</button>
    </div>
  </div>
  <p>Things cats <span class="text-danger">love:</span></p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <div class="row">
      <div class="col-xs-6">
        <label><input type="radio" name="indoor-outdoor"> Indoor</label>
      </div>
      <div class="col-xs-6">
        <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
      </div>
    </div>
    <label><input type="checkbox" name="personality"> Loving</label>
    <label><input type="checkbox" name="personality"> Lazy</label>
    <label><input type="checkbox" name="personality"> Crazy</label>
    <input type="text" placeholder="cat photo URL" required>
    <button type="submit">Submit</button>
  </form>
</div>
```

Solution

```
// solution required
```

17. Style Text Inputs as Form Controls

Description

You can add the `fa-paper-plane` Font Awesome icon by adding `<i class="fa fa-paper-plane"></i>` within your submit `button` element. Give your form's text input field a class of `form-control`. Give your form's submit button the classes `btn btn-primary`. Also give this button the Font Awesome icon of `fa-paper-plane`. All textual `<input>`, `<textarea>`, and `<select>` elements with the class `.form-control` have a width of 100%.

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  h2 {
    font-family: Lobster, Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

</style>

<div class="container-fluid">
  <div class="row">
    <div class="col-xs-8">
      <h2 class="text-primary text-center">CatPhotoApp</h2>
    </div>
    <div class="col-xs-4">
      <a href="#"><img class="img-responsive thick-green-border" src="https://bit.ly/fcc-relaxing-cat"
alt="A cute orange cat lying on its back."></a>
    </div>
  </div>
  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary"><i class="fa fa-thumbs-up"></i> Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info"><i class="fa fa-info-circle"></i> Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger"><i class="fa fa-trash"></i> Delete</button>
    </div>
  </div>
  <p>Things cats <span class="text-danger">love:</span></p>
  <ul>
    <li>cat nip</li>
    <li>laser pointers</li>
    <li>lasagna</li>
  </ul>
  <p>Top 3 things cats hate:</p>
  <ol>
    <li>flea treatment</li>
    <li>thunder</li>
    <li>other cats</li>
  </ol>
  <form action="/submit-cat-photo">
    <div class="row">
      <div class="col-xs-6">
        <label><input type="radio" name="indoor-outdoor"> Indoor</label>
      </div>
      <div class="col-xs-6">
        <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
      </div>
    </div>
    <div class="row">
      <div class="col-xs-4">
        <label><input type="checkbox" name="personality"> Loving</label>
      </div>
      <div class="col-xs-4">
        <label><input type="checkbox" name="personality"> Lazy</label>
      </div>
      <div class="col-xs-4">
        <label><input type="checkbox" name="personality"> Crazy</label>
      </div>
    </div>
```

```
      <input type="text" placeholder="cat photo URL" required>
      <button type="submit">Submit</button>
    </form>
  </div>
```

Solution

```
  // solution required
```

18. Line up Form Elements Responsively with Bootstrap

Description

Now let's get your form `input` and your submission `button` on the same line. We'll do this the same way we have previously: by using a `div` element with the class `row` , and other `div` elements within it using the `col-xs-*` class. Nest both your form's text `input` and submit `button` within a `div` with the class `row` . Nest your form's text `input` within a div with the class of `col-xs-7` . Nest your form's submit `button` in a `div` with the class `col-xs-5` . This is the last challenge we'll do for our Cat Photo App for now. We hope you've enjoyed learning Font Awesome, Bootstrap, and responsive design!

Instructions

Challenge Seed

```
<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">
<style>
  h2 {
    font-family: Lobster, Monospace;
  }

  .thick-green-border {
    border-color: green;
    border-width: 10px;
    border-style: solid;
    border-radius: 50%;
  }

</style>

<div class="container-fluid">
  <div class="row">
    <div class="col-xs-8">
      <h2 class="text-primary text-center">CatPhotoApp</h2>
    </div>
    <div class="col-xs-4">
      <a href="#"><img class="img-responsive thick-green-border" src="https://bit.ly/fcc-relaxing-cat"
alt="A cute orange cat lying on its back."></a>
    </div>
  </div>
  <img src="https://bit.ly/fcc-running-cats" class="img-responsive" alt="Three kittens running towards
the camera.">
  <div class="row">
    <div class="col-xs-4">
      <button class="btn btn-block btn-primary"><i class="fa fa-thumbs-up"></i> Like</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-info"><i class="fa fa-info-circle"></i> Info</button>
    </div>
    <div class="col-xs-4">
      <button class="btn btn-block btn-danger"><i class="fa fa-trash"></i> Delete</button>
    </div>
  </div>
  <p>Things cats <span class="text-danger">love:</span></p>
  <ul>
```

```
      <li>cat nip</li>
      <li>laser pointers</li>
      <li>lasagna</li>
    </ul>
    <p>Top 3 things cats hate:</p>
    <ol>
      <li>flea treatment</li>
      <li>thunder</li>
      <li>other cats</li>
    </ol>
    <form action="/submit-cat-photo">
      <div class="row">
        <div class="col-xs-6">
          <label><input type="radio" name="indoor-outdoor"> Indoor</label>
        </div>
        <div class="col-xs-6">
          <label><input type="radio" name="indoor-outdoor"> Outdoor</label>
        </div>
      </div>
      <div class="row">
        <div class="col-xs-4">
          <label><input type="checkbox" name="personality"> Loving</label>
        </div>
        <div class="col-xs-4">
          <label><input type="checkbox" name="personality"> Lazy</label>
        </div>
        <div class="col-xs-4">
          <label><input type="checkbox" name="personality"> Crazy</label>
        </div>
      </div>
      <input type="text" class="form-control" placeholder="cat photo URL" required>
      <button type="submit" class="btn btn-primary"><i class="fa fa-paper-plane"></i> Submit</button>
    </form>
  </div>
```

Solution

```
// solution required
```

19. Create a Bootstrap Headline

Description

Now let's build something from scratch to practice our HTML, CSS and Bootstrap skills. We'll build a jQuery playground, which we'll soon put to use in our jQuery challenges. To start with, create an `h3` element, with the text `jQuery Playground`. Color your `h3` element with the `text-primary` Bootstrap class, and center it with the `text-center` Bootstrap class.

Instructions

Challenge Seed

Solution

```
<h3 class="text-primary text-center">jQuery Playground</h3>
```

20. House our page within a Bootstrap container-fluid div

Description

Now let's make sure all the content on your page is mobile-responsive. Let's nest your `h3` element within a `div` element with the class `container-fluid` .

Instructions

Challenge Seed

```
<h3 class="text-primary text-center">jQuery Playground</h3>
```

Solution

```
// solution required
```

21. Create a Bootstrap Row

Description

Now we'll create a Bootstrap row for our inline elements. Create a `div` element below the `h3` tag, with a class of `row` .

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>

</div>
```

Solution

```
// solution required
```

22. Split Your Bootstrap Row

Description

Now that we have a Bootstrap Row, let's split it into two columns to house our elements. Create two `div` elements within your row, both with the class `col-xs-6` .

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">


  </div>
</div>
```

Solution

```
// solution required
```

23. Create Bootstrap Wells

Description

Bootstrap has a class called `well` that can create a visual sense of depth for your columns. Nest one `div` element with the class `well` within each of your `col-xs-6` `div` elements.

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">

    </div>
    <div class="col-xs-6">

    </div>
  </div>
</div>
```

Solution

```
// solution required
```

24. Add Elements within Your Bootstrap Wells

Description

Now we're several `div` elements deep on each column of our row. This is as deep as we'll need to go. Now we can add our `button` elements. Nest three `button` elements within each of your `well` `div` elements.

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <div class="well">



      </div>
    </div>
    <div class="col-xs-6">
      <div class="well">



      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

25. Apply the Default Bootstrap Button Style

Description

Bootstrap has another button class called `btn-default`. Apply both the `btn` and `btn-default` classes to each of your `button` elements.

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <div class="well">
        <button></button>
        <button></button>
        <button></button>
      </div>
    </div>
    <div class="col-xs-6">
      <div class="well">
        <button></button>
        <button></button>
        <button></button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

26. Create a Class to Target with jQuery Selectors

Description

Not every class needs to have corresponding CSS. Sometimes we create classes just for the purpose of selecting these elements more easily using jQuery. Give each of your `button` elements the class `target` .

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <div class="well">
        <button class="btn btn-default"></button>
        <button class="btn btn-default"></button>
        <button class="btn btn-default"></button>
      </div>
    </div>
    <div class="col-xs-6">
      <div class="well">
        <button class="btn btn-default"></button>
        <button class="btn btn-default"></button>
        <button class="btn btn-default"></button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

27. Add id Attributes to Bootstrap Elements

Description

Recall that in addition to class attributes, you can give each of your elements an `id` attribute. Each id must be unique to a specific element and used only once per page. Let's give a unique id to each of our `div` elements of class `well` . Remember that you can give an element an id like this: `<div class="well" id="center-well">` Give the well on the left the id of `left-well` . Give the well on the right the id of `right-well` .

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <div class="well">
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
      </div>
    </div>
```

```
      <div class="col-xs-6">
        <div class="well">
          <button class="btn btn-default target"></button>
          <button class="btn btn-default target"></button>
          <button class="btn btn-default target"></button>
        </div>
      </div>
    </div>
  </div>
```

Solution

```
// solution required
```

28. Label Bootstrap Wells

Description

For the sake of clarity, let's label both of our wells with their ids. Above your left-well, inside its `col-xs-6 div` element, add a `h4` element with the text `#left-well`. Above your right-well, inside its `col-xs-6 div` element, add a `h4` element with the text `#right-well`.

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">

      <div class="well" id="left-well">
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
      </div>
    </div>
    <div class="col-xs-6">

      <div class="well" id="right-well">
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

29. Give Each Element a Unique id

Description

We will also want to be able to use jQuery to target each button by its unique id. Give each of your buttons a unique id, starting with `target1` and ending with `target6`. Make sure that `target1` to `target3` are in `#left-well`, and `target4` to `target6` are in `#right-well`.

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
        <button class="btn btn-default target"></button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

30. Label Bootstrap Buttons

Description

Just like we labeled our wells, we want to label our buttons. Give each of your `button` elements text that corresponds to its `id`'s selector.

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1"></button>
        <button class="btn btn-default target" id="target2"></button>
        <button class="btn btn-default target" id="target3"></button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4"></button>
        <button class="btn btn-default target" id="target5"></button>
        <button class="btn btn-default target" id="target6"></button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

31. Use Comments to Clarify Code

Description

When we start using jQuery, we will modify HTML elements without needing to actually change them in HTML. Let's make sure that everyone knows they shouldn't actually modify any of this code directly. Remember that you can start a comment with `<!--` and end a comment with `-->` Add a comment at the top of your HTML that says `Only change code above this line.`

Instructions

Challenge Seed

```
<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

jQuery

1. Learn How Script Tags and Document Ready Work

Description

Now we're ready to learn jQuery, the most popular JavaScript tool of all time. Before we can start using jQuery, we need to add some things to our HTML. First, add a `script` element at the top of your page. Be sure to close it on the following line. Your browser will run any JavaScript inside a `script` element, including jQuery. Inside your `script` element, add this code: `$(document).ready(function() {` to your `script` . Then close it on the following line (still inside your `script` element) with: `});` We'll learn more about `functions` later. The important thing to know is that code you put inside this `function` will run as soon as your browser has loaded your page. This is important because without your `document ready function` , your code may run before your HTML is rendered, which would cause bugs.

Instructions

Challenge Seed

```
<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

2. Target HTML Elements with Selectors Using jQuery

Description

Now we have a `document ready function` . Now let's write our first jQuery statement. All jQuery functions start with a `$` , usually referred to as a `dollar sign operator` , or as `bling` . jQuery often selects an HTML element with a `selector` , then does something to that element. For example, let's make all of your `button` elements bounce. Just add this code inside your document ready function: `$("button").addClass("animated bounce");` Note that we've already included both the jQuery library and the Animate.css library in the background so that you can use them in the editor. So you are using jQuery to apply the Animate.css `bounce` class to your `button` elements.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {

  });
```

```
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

3. Target Elements by Class Using jQuery

Description

You see how we made all of your `button` elements bounce? We selected them with `$("button")`, then we added some CSS classes to them with `.addClass("animated bounce");`. You just used jQuery's `.addClass()` function, which allows you to add classes to elements. First, let's target your `div` elements with the class `well` by using the `$(".well")` selector. Note that, just like with CSS declarations, you type a `.` before the class's name. Then use jQuery's `.addClass()` function to add the classes `animated` and `shake`. For example, you could make all the elements with the class `text-primary` shake by adding the following to your `document ready function`: `$(".text-primary").addClass("animated shake");`

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("button").addClass("animated bounce");
  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

4. Target Elements by id Using jQuery

Description

You can also target elements by their id attributes. First target your `button` element with the id `target3` by using the `$("#target3")` selector. Note that, just like with CSS declarations, you type a `#` before the id's name. Then use jQuery's `.addClass()` function to add the classes `animated` and `fadeOut`. Here's how you'd make the `button` element with the id `target6` fade out: `$("#target6").addClass("animated fadeOut")`.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("button").addClass("animated bounce");
    $(".well").addClass("animated shake");

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

5. Delete Your jQuery Functions

Description

These animations were cool at first, but now they're getting kind of distracting. Delete all three of these jQuery functions from your `document ready function`, but leave your `document ready function` itself intact.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("button").addClass("animated bounce");
    $(".well").addClass("animated shake");
    $("#target3").addClass("animated fadeOut");

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

6. Target the Same Element with Multiple jQuery Selectors

Description

Now you know three ways of targeting elements: by type: `$("button")`, by class: `$(".btn")`, and by id `$("#target1")`. Although it is possible to add multiple classes in a single `.addClass()` call, let's add them to the

same element in *three separate ways*. Using `.addClass()`, add only one class at a time to the same element, three different ways: Add the `animated` class to all elements with type `button`. Add the `shake` class to all the buttons with class `.btn`. Add the `btn-primary` class to the button with id `#target1`. **Note**
You should only be targeting one element and adding only one class at a time. Altogether, your three individual selectors will end up adding the three classes `shake`, `animated`, and `btn-primary` to `#target1`.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

7. Remove Classes from an Element with jQuery

Description

In the same way you can add classes to an element with jQuery's `addClass()` function, you can remove them with jQuery's `removeClass()` function. Here's how you would do this for a specific button: `$("#target2").removeClass("btn-default");` Let's remove the `btn-default` class from all of our `button` elements.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("button").addClass("animated bounce");
    $(".well").addClass("animated shake");
    $("#target3").addClass("animated fadeOut");

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

8. Change the CSS of an Element Using jQuery

Description

We can also change the CSS of an HTML element directly with jQuery. jQuery has a function called `.css()` that allows you to change the CSS of an element. Here's how we would change its color to blue: `$("#target1").css("color", "blue");` This is slightly different from a normal CSS declaration, because the CSS property and its value are in quotes, and separated with a comma instead of a colon. Delete your jQuery selectors, leaving an empty `document ready function`. Select `target1` and change its color to red.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("button").addClass("animated bounce");
    $(".well").addClass("animated shake");
    $("#target3").addClass("animated fadeOut");
    $("button").removeClass("btn-default");

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

9. Disable an Element Using jQuery

Description

You can also change the non-CSS properties of HTML elements with jQuery. For example, you can disable buttons. When you disable a button, it will become grayed-out and can no longer be clicked. jQuery has a function called `.prop()` that allows you to adjust the properties of elements. Here's how you would disable all buttons: `$("button").prop("disabled", true);` Disable only the `target1` button.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

10. Change Text Inside an Element Using jQuery

Description

Using jQuery, you can change the text between the start and end tags of an element. You can even change HTML markup. jQuery has a function called `.html()` that lets you add HTML tags and text within an element. Any content previously within the element will be completely replaced with the content you provide using this function. Here's how you would rewrite and emphasize the text of our heading: `$("h3").html("<em>jQuery Playground</em>");` jQuery also has a similar function called `.text()` that only alters text without adding tags. In other words, this function will not evaluate any HTML tags passed to it, but will instead treat it as the text you want to replace the existing content with. Change the button with id `target4` by emphasizing its text. Check this link to know more on the difference between `<i>` and `<em>` and their uses. Note that while the `<i>` tag has traditionally been used to emphasize text, it has since been coopted for use as a tag for icons. The `<em>` tag is now widely accepted as the tag for emphasis. Either will work for this challenge.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");
    $("#target4").html('<em>#target4</em>');
  });
```

```
</script>

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

11. Remove an Element Using jQuery

Description

Now let's remove an HTML element from your page using jQuery. jQuery has a function called `.remove()` that will remove an HTML element entirely Remove element `target4` from the page by using the `.remove()` function.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");
    $("#target1").prop("disabled", true);

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

12. Use appendTo to Move Elements with jQuery

Description

Now let's try moving elements from one `div` to another. jQuery has a function called `appendTo()` that allows you to select HTML elements and append them to another element. For example, if we wanted to move `target4` from our right well to our left well, we would use: `$("#target4").appendTo("#left-well");` Move your `target2` element from your `left-well` to your `right-well` .

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");
    $("#target1").prop("disabled", true);
    $("#target4").remove();

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

13. Clone an Element Using jQuery

Description

In addition to moving elements, you can also copy them from one place to another. jQuery has a function called `clone()` that makes a copy of an element. For example, if we wanted to copy `target2` from our `left-well` to our `right-well`, we would use: `$("#target2").clone().appendTo("#right-well");` Did you notice this involves sticking two jQuery functions together? This is called `function chaining` and it's a convenient way to get things done with jQuery. Clone your `target5` element and append it to your `left-well`.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");
    $("#target1").prop("disabled", true);
    $("#target4").remove();
    $("#target2").appendTo("#right-well");

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

14. Target the Parent of an Element Using jQuery

Description

Every HTML element has a `parent` element from which it `inherits` properties. For example, your `jQuery Playground` `h3` element has the parent element of `<div class="container-fluid">`, which itself has the parent `body`. jQuery has a function called `parent()` that allows you to access the parent of whichever element you've selected. Here's an example of how you would use the `parent()` function if you wanted to give the parent element of the `left-well` element a background color of blue: `$("#left-well").parent().css("background-color", "blue")` Give the parent of the `#target1` element a background-color of red.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");
    $("#target1").prop("disabled", true);
    $("#target4").remove();
    $("#target2").appendTo("#right-well");
    $("#target5").clone().appendTo("#left-well");

  });
</script>

<!-- Only change code above this line. -->

<body>
  <div class="container-fluid">
    <h3 class="text-primary text-center">jQuery Playground</h3>
    <div class="row">
      <div class="col-xs-6">
        <h4>#left-well</h4>
        <div class="well" id="left-well">
          <button class="btn btn-default target" id="target1">#target1</button>
          <button class="btn btn-default target" id="target2">#target2</button>
          <button class="btn btn-default target" id="target3">#target3</button>
        </div>
      </div>
      <div class="col-xs-6">
        <h4>#right-well</h4>
        <div class="well" id="right-well">
          <button class="btn btn-default target" id="target4">#target4</button>
          <button class="btn btn-default target" id="target5">#target5</button>
          <button class="btn btn-default target" id="target6">#target6</button>
        </div>
      </div>
    </div>
  </div>
</body>
```

Solution

```
// solution required
```

15. Target the Children of an Element Using jQuery

Description

When HTML elements are placed one level below another they are called `children` of that element. For example, the button elements in this challenge with the text "#target1", "#target2", and "#target3" are all `children` of the `<div class="well" id="left-well">` element. jQuery has a function called `children()` that allows you to access the children of whichever element you've selected. Here's an example of how you would use the `children()` function to give the children of your `left-well` element the color `blue` : `$("#left-well").children().css("color", "blue")`

Instructions

Give all the children of your `right-well` element the color orange.

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");
    $("#target1").prop("disabled", true);
```

```
      $("#target4").remove();
      $("#target2").appendTo("#right-well");
      $("#target5").clone().appendTo("#left-well");
      $("#target1").parent().css("background-color", "red");

    });
  </script>

  <!-- Only change code above this line. -->

  <div class="container-fluid">
    <h3 class="text-primary text-center">jQuery Playground</h3>
    <div class="row">
      <div class="col-xs-6">
        <h4>#left-well</h4>
        <div class="well" id="left-well">
          <button class="btn btn-default target" id="target1">#target1</button>
          <button class="btn btn-default target" id="target2">#target2</button>
          <button class="btn btn-default target" id="target3">#target3</button>
        </div>
      </div>
      <div class="col-xs-6">
        <h4>#right-well</h4>
        <div class="well" id="right-well">
          <button class="btn btn-default target" id="target4">#target4</button>
          <button class="btn btn-default target" id="target5">#target5</button>
          <button class="btn btn-default target" id="target6">#target6</button>
        </div>
      </div>
    </div>
  </div>
```

Solution

```
// solution required
```

16. Target a Specific Child of an Element Using jQuery

Description

You've seen why id attributes are so convenient for targeting with jQuery selectors. But you won't always have such neat ids to work with. Fortunately, jQuery has some other tricks for targeting the right elements. jQuery uses CSS Selectors to target elements. The `target:nth-child(n)` CSS selector allows you to select all the nth elements with the target class or element type. Here's how you would give the third element in each well the bounce class: `$(".target:nth-child(3)").addClass("animated bounce");` Make the second child in each of your well elements bounce. You must select the elements' children with the `target` class.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");
    $("#target1").prop("disabled", true);
    $("#target4").remove();
    $("#target2").appendTo("#right-well");
    $("#target5").clone().appendTo("#left-well");
    $("#target1").parent().css("background-color", "red");
    $("#right-well").children().css("color", "orange");

  });
</script>
```

```
<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

17. Target Even Elements Using jQuery

Description

You can also target elements based on their positions using `:odd` or `:even` selectors. Note that jQuery is zero-indexed which means the first element in a selection has a position of 0. This can be a little confusing as, counter-intuitively, `:odd` selects the second element (position 1), fourth element (position 3), and so on. Here's how you would target all the odd elements with class `target` and give them classes: `$(".target:odd").addClass("animated shake");` Try selecting all the even `target` elements and giving them the classes of `animated` and `shake` . Remember that **even** refers to the position of elements with a zero-based system in mind.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");
    $("#target1").prop("disabled", true);
    $("#target4").remove();
    $("#target2").appendTo("#right-well");
    $("#target5").clone().appendTo("#left-well");
    $("#target1").parent().css("background-color", "red");
    $("#right-well").children().css("color", "orange");
    $("#left-well").children().css("color", "green");
    $(".target:nth-child(2)").addClass("animated bounce");

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
```

```
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

18. Use jQuery to Modify the Entire Page

Description

We're done playing with our jQuery playground. Let's tear it down! jQuery can target the `body` element as well. Here's how we would make the entire body fade out: `$("body").addClass("animated fadeOut");` But let's do something more dramatic. Add the classes `animated` and `hinge` to your `body` element.

Instructions

Challenge Seed

```
<script>
  $(document).ready(function() {
    $("#target1").css("color", "red");
    $("#target1").prop("disabled", true);
    $("#target4").remove();
    $("#target2").appendTo("#right-well");
    $("#target5").clone().appendTo("#left-well");
    $("#target1").parent().css("background-color", "red");
    $("#right-well").children().css("color", "orange");
    $("#left-well").children().css("color", "green");
    $(".target:nth-child(2)").addClass("animated bounce");
    $(".target:even").addClass("animated shake");

  });
</script>

<!-- Only change code above this line. -->

<div class="container-fluid">
  <h3 class="text-primary text-center">jQuery Playground</h3>
  <div class="row">
    <div class="col-xs-6">
      <h4>#left-well</h4>
      <div class="well" id="left-well">
        <button class="btn btn-default target" id="target1">#target1</button>
        <button class="btn btn-default target" id="target2">#target2</button>
        <button class="btn btn-default target" id="target3">#target3</button>
      </div>
    </div>
    <div class="col-xs-6">
      <h4>#right-well</h4>
      <div class="well" id="right-well">
        <button class="btn btn-default target" id="target4">#target4</button>
        <button class="btn btn-default target" id="target5">#target5</button>
        <button class="btn btn-default target" id="target6">#target6</button>
      </div>
    </div>
  </div>
</div>
```

Solution

```
// solution required
```

Sass

1. Store Data with Sass Variables

Description

One feature of Sass that's different than CSS is it uses variables. They are declared and set to store data, similar to JavaScript. In JavaScript, variables are defined using the `let` and `const` keywords. In Sass, variables start with a `$` followed by the variable name. Here are a couple examples:

> $main-fonts: Arial, sans-serif;
> $headings-color: green;
>
> //To use variables:
> h1 {
>   font-family: $main-fonts;
>   color: $headings-color;
> }

One example where variables are useful is when a number of elements need to be the same color. If that color is changed, the only place to edit the code is the variable value.

Instructions

Create a variable `$text-color` and set it to red. Then change the value of the `color` property for the `.blog-post` and `h2` to the `$text-color` variable.

Challenge Seed

```
<style type='text/sass'>


  .header{
    text-align: center;
  }
  .blog-post, h2 {
    color: red;
  }
</style>

<h1 class="header">Learn Sass</h1>
<div class="blog-post">
  <h2>Some random title</h2>
  <p>This is a paragraph with some random text in it</p>
</div>
<div class="blog-post">
  <h2>Header #2</h2>
  <p>Here is some more random text.</p>
```

```
</div>
<div class="blog-post">
  <h2>Here is another header</h2>
  <p>Even more random text within a paragraph</p>
</div>
```

Solution

```
// solution required
```

2. Nest CSS with Sass

Description

Sass allows `nesting` of CSS rules, which is a useful way of organizing a style sheet. Normally, each element is targeted on a different line to style it, like so:

> nav {
> background-color: red;
> }
>
> nav ul {
> list-style: none;
> }
>
> nav ul li {
> display: inline-block;
> }

For a large project, the CSS file will have many lines and rules. This is where `nesting` can help organize your code by placing child style rules within the respective parent elements:

> nav {
> background-color: red;
>
> ul {
>   list-style: none;
>
>   li {
>   display: inline-block;
>   }
> }
> }

Instructions

Use the `nesting` technique shown above to re-organize the CSS rules for both children of `.blog-post` element. For testing purposes, the `h1` should come before the `p` element.

Challenge Seed

```
<style type='text/sass'>
  .blog-post {

  }
  h1 {
    text-align: center;
    color: blue;
  }
  p {
    font-size: 20px;
  }
</style>

<div class="blog-post">
  <h1>Blog Title</h1>
  <p>This is a paragraph</p>
</div>
```

Solution

```
// solution required
```

3. Create Reusable CSS with Mixins

Description

In Sass, a `mixin` is a group of CSS declarations that can be reused throughout the style sheet. Newer CSS features take time before they are fully adopted and ready to use in all browsers. As features are added to browsers, CSS rules using them may need vendor prefixes. Consider "box-shadow":

> div {
>   -webkit-box-shadow: 0px 0px 4px #fff;
>   -moz-box-shadow: 0px 0px 4px #fff;
>   -ms-box-shadow: 0px 0px 4px #fff;
>   box-shadow: 0px 0px 4px #fff;
> }

It's a lot of typing to re-write this rule for all the elements that have a `box-shadow`, or to change each value to test different effects. `Mixins` are like functions for CSS. Here is how to write one:

> @mixin box-shadow($x, $y, $blur, $c){
>   -webkit-box-shadow: $x, $y, $blur, $c;
>   -moz-box-shadow: $x, $y, $blur, $c;
>   -ms-box-shadow: $x, $y, $blur, $c;
>   box-shadow: $x, $y, $blur, $c;
> }

The definition starts with `@mixin` followed by a custom name. The parameters (the `$x`, `$y`, `$blur`, and `$c` in the example above) are optional. Now any time a `box-shadow` rule is needed, only a single line calling the `mixin` replaces having to type all the vendor prefixes. A `mixin` is called with the `@include` directive:

> div {
>   @include box-shadow(0px, 0px, 4px, #fff);
> }

Instructions

Write a `mixin` for `border-radius` and give it a `$radius` parameter. It should use all the vendor prefixes from the example. Then use the `border-radius` `mixin` to give the `#awesome` element a border radius of 15px.

Challenge Seed

```
<style type='text/sass'>



  #awesome {
    width: 150px;
    height: 150px;
    background-color: green;

  }
</style>
```

```
<div id="awesome"></div>
```

Solution

```
// solution required
```

4. Use @if and @else to Add Logic To Your Styles

Description

The `@if` directive in Sass is useful to test for a specific case - it works just like the `if` statement in JavaScript.

> @mixin make-bold($bool) {
> @if $bool == true {
>   font-weight: bold;
> }
> }

And just like in JavaScript, `@else if` and `@else` test for more conditions:

> @mixin text-effect($val) {
> @if $val == danger {
>   color: red;
> }
> @else if $val == alert {
>   color: yellow;
> }
> @else if $val == success {
>   color: green;
> }
> @else {
>   color: black;
> }
> }

Instructions

Create a `mixin` called `border-stroke` that takes a parameter `$val`. The `mixin` should check for the following conditions using `@if`, `@else if`, and `@else`:

> light - 1px solid black
> medium - 3px solid black
> heavy - 6px solid black

If `$val` is not `light`, `medium`, or `heavy`, the border should be set to `none`.

Challenge Seed

```
<style type='text/sass'>



  #box {
    width: 150px;
    height: 150px;
    background-color: red;
    @include border-stroke(medium);
  }
</style>

<div id="box"></div>
```

Solution

```
// solution required
```

5. Use @for to Create a Sass Loop

Description

The `@for` directive adds styles in a loop, very similar to a `for` loop in JavaScript. `@for` is used in two ways: "start through end" or "start to end". The main difference is that the "start **to** end" *excludes* the end number as part of the count, and "start **through** end" *includes* the end number as part of the count. Here's a start **through** end example:

> @for $i from 1 through 12 {
>   .col-#{$i} { width: 100%/12 * $i; }
> }

The `#{$i}` part is the syntax to combine a variable (`i`) with text to make a string. When the Sass file is converted to CSS, it looks like this:

> .col-1 {
>   width: 8.33333%;
> }
>
> .col-2 {
>   width: 16.66667%;
> }
>
> ...
>
> .col-12 {
>   width: 100%;
> }

This is a powerful way to create a grid layout. Now you have twelve options for column widths available as CSS classes.

Instructions

Write a `@for` directive that takes a variable `$j` that goes from 1 **to** 6. It should create 5 classes called `.text-1` to `.text-5` where each has a `font-size` set to 10px multiplied by the index.

Challenge Seed

```
<style type='text/sass'>



</style>

<p class="text-1">Hello</p>
<p class="text-2">Hello</p>
<p class="text-3">Hello</p>
<p class="text-4">Hello</p>
<p class="text-5">Hello</p>
```

Solution

```
<style type='text/sass'>

@for $i from 1 through 5 {
  .text-#{$i} { font-size: 10px * $i; }
}

</style>

<p class="text-1">Hello</p>
<p class="text-2">Hello</p>
<p class="text-3">Hello</p>
<p class="text-4">Hello</p>
<p class="text-5">Hello</p>



<style type='text/sass'>

@for $i from 1 to 6 {
  .text-#{$i} { font-size: 10px * $i; }
}

</style>

<p class="text-1">Hello</p>
<p class="text-2">Hello</p>
<p class="text-3">Hello</p>
<p class="text-4">Hello</p>
<p class="text-5">Hello</p>
```

6. Use @each to Map Over Items in a List

Description

The last challenge showed how the `@for` directive uses a starting and ending value to loop a certain number of times. Sass also offers the `@each` directive which loops over each item in a list or map. On each iteration, the variable gets assigned to the current value from the list or map.

> @each $color in blue, red, green {
>   .#{$color}-text {color: $color;}
> }

A map has slightly different syntax. Here's an example:

> $colors: (color1: blue, color2: red, color3: green);
>
> @each $key, $color in $colors {
>   .#{$color}-text {color: $color;}
> }

Note that the `$key` variable is needed to reference the keys in the map. Otherwise, the compiled CSS would have `color1`, `color2`... in it. Both of the above code examples are converted into the following CSS:

> .blue-text {
>   color: blue;
> }
>
> .red-text {
>   color: red;
> }
>
> .green-text {
>   color: green;
> }

Instructions

Write an `@each` directive that goes through a list: `blue`, `black`, `red` and assigns each variable to a `.color-bg` class, where the "color" part changes for each item. Each class should set the `background-color` the respective color.

Challenge Seed

```
<style type='text/sass'>



  div {
    height: 200px;
    width: 200px;
  }
</style>

<div class="blue-bg"></div>
<div class="black-bg"></div>
<div class="red-bg"></div>
```

Solution

The solution requires using the $color variable twice: once for the class name and once for setting the background color. You can use either the list or map data type.

List Data type

```
<style type='text/sass'>

  @each $color in blue, black, red {
    .#{$color}-bg {background-color: $color;}
  }

  div {
    height: 200px;
    width: 200px;
  }
</style>

<div class="blue-bg"></div>
<div class="black-bg"></div>
<div class="red-bg"></div>
```

Map Data type

```
<style type='text/sass'>

  $colors: (color1: blue, color2: black, color3: red);

  @each $key, $color in $colors {
    .#{$color}-bg {background-color: $color;}
  }

  div {
    height: 200px;
    width: 200px;
  }
</style>

<div class="blue-bg"></div>
<div class="black-bg"></div>
<div class="red-bg"></div>
```

7. Apply a Style Until a Condition is Met with @while

Description

The `@while` directive is an option with similar functionality to the JavaScript `while` loop. It creates CSS rules until a condition is met. The `@for` challenge gave an example to create a simple grid system. This can also work with `@while`.

> $x: 1;
> @while $x < 13 {
>   .col-#{$x} { width: 100%/12 * $x;}
> $x: $x + 1;
> }

First, define a variable `$x` and set it to 1. Next, use the `@while` directive to create the grid system *while* `$x` is less than 13. After setting the CSS rule for `width`, `$x` is incremented by 1 to avoid an infinite loop.

Instructions

Use `@while` to create a series of classes with different `font-sizes`. There should be 10 different classes from `text-1` to `text-10`. Then set `font-size` to 5px multiplied by the current index number. Make sure to avoid an infinite loop!

Challenge Seed

```html
<style type='text/sass'>



</style>

<p class="text-1">Hello</p>
<p class="text-2">Hello</p>
<p class="text-3">Hello</p>
<p class="text-4">Hello</p>
<p class="text-5">Hello</p>
<p class="text-6">Hello</p>
<p class="text-7">Hello</p>
<p class="text-8">Hello</p>
<p class="text-9">Hello</p>
<p class="text-10">Hello</p>
```

Solution

```
// solution required
```

8. Split Your Styles into Smaller Chunks with Partials

Description

`Partials` in Sass are separate files that hold segments of CSS code. These are imported and used in other Sass files. This is a great way to group similar code into a module to keep it organized. Names for `partials` start with the underscore (_) character, which tells Sass it is a small segment of CSS and not to convert it into a CSS file. Also, Sass files end with the `.scss` file extension. To bring the code in the `partial` into another Sass file, use the `@import` directive. For example, if all your `mixins` are saved in a `partial` named "_mixins.scss", and they are needed in the "main.scss" file, this is how to use them in the main file:

> // In the main.scss file
>
> @import 'mixins'

Note that the underscore is not needed in the `import` statement - Sass understands it is a `partial`. Once a `partial` is imported into a file, all variables, `mixins`, and other code are available to use.

Instructions

Write an `@import` statement to import a `partial` named `_variables.scss` into the main.scss file.

Challenge Seed

```
// The main.scss file
```

Solution

```
// solution required
```

9. Extend One Set of CSS Styles to Another Element

Description

Sass has a feature called `extend` that makes it easy to borrow the CSS rules from one element and build upon them in another. For example, the below block of CSS rules style a `.panel` class. It has a `background-color`, `height` and `border`.

> .panel{
>   background-color: red;
>   height: 70px;
>   border: 2px solid green;
> }

Now you want another panel called `.big-panel`. It has the same base properties as `.panel`, but also needs a `width` and `font-size`. It's possible to copy and paste the initial CSS rules from `.panel`, but the code becomes repetitive as you add more types of panels. The `extend` directive is a simple way to reuse the rules written for one element, then add more for another:

> .big-panel{
>   @extend .panel;
>   width: 150px;
>   font-size: 2em;
> }

The `.big-panel` will have the same properties as `.panel` in addition to the new styles.

Instructions

Make a class `.info-important` that extends `.info` and also has a `background-color` set to magenta.

Challenge Seed

```
<style type='text/sass'>
  h3{
    text-align: center;
  }
  .info{
    width: 200px;
    border: 1px solid black;
    margin: 0 auto;
  }




</style>
<h3>Posts</h3>
```

```
<div class="info-important">
  <p>This is an important post. It should extend the class ".info" and have its own CSS styles.</p>
</div>

<div class="info">
  <p>This is a simple post. It has basic styling and can be extended for other uses.</p>
</div>
```

Solution

```
// solution required
```

React

1. Create a Simple JSX Element

Description

Intro: React is an Open Source view library created and maintained by Facebook. It's a great tool to render the User Interface (UI) of modern web applications. React uses a syntax extension of JavaScript called JSX that allows you to write HTML directly within JavaScript. This has several benefits. It lets you use the full programmatic power of JavaScript within HTML, and helps to keep your code readable. For the most part, JSX is similar to the HTML that you have already learned, however there are a few key differences that will be covered throughout these challenges. For instance, because JSX is a syntactic extension of JavaScript, you can actually write JavaScript directly within JSX. To do this, you simply include the code you want to be treated as JavaScript within curly braces: `{ 'this is treated as JavaScript code' }`. Keep this in mind, since it's used in several future challenges. However, because JSX is not valid JavaScript, JSX code must be compiled into JavaScript. The transpiler Babel is a popular tool for this process. For your convenience, it's already added behind the scenes for these challenges. If you happen to write syntactically invalid JSX, you will see the first test in these challenges fail. It's worth noting that under the hood the challenges are calling `ReactDOM.render(JSX, document.getElementById('root'))`. This function call is what places your JSX into React's own lightweight representation of the DOM. React then uses snapshots of its own DOM to optimize updating only specific parts of the actual DOM.

Instructions

Instructions: The current code uses JSX to assign a `div` element to the constant `JSX`. Replace the `div` with an `h1` element and add the text `Hello JSX!` inside it.

Challenge Seed

```
const JSX = <div></div>;
```

After Test

```
ReactDOM.render(JSX, document.getElementById('root'))
```

Solution

```
const JSX = <h1>Hello JSX!</h1>;
```

2. Create a Complex JSX Element

Description

The last challenge was a simple example of JSX, but JSX can represent more complex HTML as well. One important thing to know about nested JSX is that it must return a single element. This one parent element would wrap all of the other levels of nested elements. For instance, several JSX elements written as siblings with no parent wrapper element will not transpile. Here's an example: **Valid JSX:**

> <div>
>   <p>Paragraph One</p>
>   <p>Paragraph Two</p>
>   <p>Paragraph Three</p>
> </div>

Invalid JSX:

> <p>Paragraph One</p>
> <p>Paragraph Two</p>
> <p>Paragraph Three</p>

Instructions

Define a new constant `JSX` that renders a `div` which contains the following elements in order: An `h1` , a `p` , and an unordered list that contains three `li` items. You can include any text you want within each element. **Note:** When rendering multiple elements like this, you can wrap them all in parentheses, but it's not strictly required. Also notice this challenge uses a `div` tag to wrap all the child elements within a single parent element. If you remove the `div` , the JSX will no longer transpile. Keep this in mind, since it will also apply when you return JSX elements in React components.

Challenge Seed

```
// write your code here
```

After Test

```
ReactDOM.render(JSX, document.getElementById('root'))
```

Solution

```
const JSX = (
<div>
  <h1>Hello JSX!</h1>
  <p>Some info</p>
  <ul>
    <li>An item</li>
    <li>Another item</li>
    <li>A third item</li>
  </ul>
</div>);
```

3. Add Comments in JSX

Description

JSX is a syntax that gets compiled into valid JavaScript. Sometimes, for readability, you might need to add comments to your code. Like most programming languages, JSX has its own way to do this. To put comments inside JSX, you use the syntax `{/* */}` to wrap around the comment text.

Instructions

The code editor has a JSX element similar to what you created in the last challenge. Add a comment somewhere within the provided `div` element, without modifying the existing `h1` or `p` elements.

Challenge Seed

```
const JSX = (
  <div>
    <h1>This is a block of JSX</h1>
    <p>Here's a subtitle</p>
  </div>
);
```

After Test

```
ReactDOM.render(JSX, document.getElementById('root'))
```

Solution

```
const JSX = (
<div>
  <h1>This is a block of JSX</h1>
  { /* this is a JSX comment */ }
  <p>Here's a subtitle</p>
</div>);
```

4. Render HTML Elements to the DOM

Description

So far, you've learned that JSX is a convenient tool to write readable HTML within JavaScript. With React, we can render this JSX directly to the HTML DOM using React's rendering API known as ReactDOM. ReactDOM offers a simple method to render React elements to the DOM which looks like this: `ReactDOM.render(componentToRender, targetNode)`, where the first argument is the React element or component that you want to render, and the second argument is the DOM node that you want to render the component to. As you would expect, `ReactDOM.render()` must be called after the JSX element declarations, just like how you must declare variables before using them.

Instructions

The code editor has a simple JSX component. Use the `ReactDOM.render()` method to render this component to the page. You can pass defined JSX elements directly in as the first argument and use `document.getElementById()` to select the DOM node to render them to. There is a `div` with `id='challenge-node'` available for you to use. Make sure you don't change the `JSX` constant.

Challenge Seed

```
const JSX = (
  <div>
    <h1>Hello World</h1>
    <p>Lets render this to the DOM</p>
  </div>
);
// change code below this line
```

Solution

```
const JSX = (
<div>
  <h1>Hello World</h1>
  <p>Lets render this to the DOM</p>
</div>
);
// change code below this line
ReactDOM.render(JSX, document.getElementById('challenge-node'));
```

5. Define an HTML Class in JSX

Description

Now that you're getting comfortable writing JSX, you may be wondering how it differs from HTML. So far, it may seem that HTML and JSX are exactly the same. One key difference in JSX is that you can no longer use the word `class` to define HTML classes. This is because `class` is a reserved word in JavaScript. Instead, JSX uses `className` . In fact, the naming convention for all HTML attributes and event references in JSX become camelCase. For example, a click event in JSX is `onClick` , instead of `onclick` . Likewise, `onchange` becomes `onChange` . While this is a subtle difference, it is an important one to keep in mind moving forward.

Instructions

Apply a class of `myDiv` to the `div` provided in the JSX code.

Challenge Seed

```
const JSX = (
  <div>
    <h1>Add a class to this div</h1>
  </div>
);
```

After Test

```
ReactDOM.render(JSX, document.getElementById('root'))
```

Solution

```
const JSX = (
<div className = 'myDiv'>
  <h1>Add a class to this div</h1>
</div>);
```

6. Learn About Self-Closing JSX Tags

Description

So far, you've seen how JSX differs from HTML in a key way with the use of `className` vs. `class` for defining HTML classes. Another important way in which JSX differs from HTML is in the idea of the self-closing tag. In HTML, almost all tags have both an opening and closing tag: `<div></div>` ; the closing tag always has a forward slash before the tag name that you are closing. However, there are special instances in HTML called "self-closing tags", or tags that don't require both an opening and closing tag before another tag can start. For example the line-break tag can be written as `<br>` or as `<br />` , but should never be written as `<br></br>` , since it doesn't contain any content. In JSX, the rules are a little different. Any JSX element can be written with a self-closing tag, and every element must be closed.

The line-break tag, for example, must always be written as `<br />` in order to be valid JSX that can be transpiled. A `<div>` , on the other hand, can be written as `<div />` or `<div></div>` . The difference is that in the first syntax version there is no way to include anything in the `<div />` . You will see in later challenges that this syntax is useful when rendering React components.

Instructions

Fix the errors in the code editor so that it is valid JSX and successfully transpiles. Make sure you don't change any of the content - you only need to close tags where they are needed.

Challenge Seed

```
const JSX = (
  <div>
    {/* remove comment and change code below this line
    <h2>Welcome to React!</h2> <br >
    <p>Be sure to close all tags!</p>
    <hr >
    remove comment and change code above this line */}
  </div>
);
```

After Test

```
ReactDOM.render(JSX, document.getElementById('root'))
```

Solution

```
const JSX = (
<div>
  {/* change code below this line */}
  <h2>Welcome to React!</h2> <br />
  <p>Be sure to close all tags!</p>
  <hr />
  {/* change code above this line */}
</div>
);
```

7. Create a Stateless Functional Component

Description

Components are the core of React. Everything in React is a component and here you will learn how to create one. There are two ways to create a React component. The first way is to use a JavaScript function. Defining a component in this way creates a *stateless functional component*. The concept of state in an application will be covered in later challenges. For now, think of a stateless component as one that can receive data and render it, but does not manage or track changes to that data. (We'll cover the second way to create a React component in the next challenge.) To create a component with a function, you simply write a JavaScript function that returns either JSX or `null` . One important thing to note is that React requires your function name to begin with a capital letter. Here's an example of a stateless functional component that assigns an HTML class in JSX:

> // After being transpiled, the <div> will have a CSS class of 'customClass'
> const DemoComponent = function() {
>   return (
>   <div className='customClass' />
>   );
> };

Because a JSX component represents HTML, you could put several components together to create a more complex HTML page. This is one of the key advantages of the component architecture React provides. It allows you to compose your UI from many separate, isolated components. This makes it easier to build and maintain complex user interfaces.

Instructions

The code editor has a function called `MyComponent` . Complete this function so it returns a single `div` element which contains some string of text. **Note:** The text is considered a child of the `div` element, so you will not be able to use a self-closing tag.

Challenge Seed

```
const MyComponent = function() {
  // change code below this line



  // change code above this line
}
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
const MyComponent = function() {
  // change code below this line
  return (
    <div>
      Demo Solution
    </div>
  );
  // change code above this line
}
```

8. Create a React Component

Description

The other way to define a React component is with the ES6 `class` syntax. In the following example, `Kitten` extends `React.Component` :

> class Kitten extends React.Component {
> constructor(props) {
>   super(props);
> }
>
> render() {
>   return (
>   <h1>Hi</h1>
>   );
> }
> }

This creates an ES6 class `Kitten` which extends the `React.Component` class. So the `Kitten` class now has access to many useful React features, such as local state and lifecycle hooks. Don't worry if you aren't familiar with these terms yet, they will be covered in greater detail in later challenges. Also notice the `Kitten` class has a `constructor` defined within it that calls `super()` . It uses `super()` to call the constructor of the parent class, in this case `React.Component` .

The constructor is a special method used during the initialization of objects that are created with the `class` keyword. It is best practice to call a component's `constructor` with `super`, and pass `props` to both. This makes sure the component is initialized properly. For now, know that it is standard for this code to be included. Soon you will see other uses for the constructor as well as `props`.

Instructions

`MyComponent` is defined in the code editor using class syntax. Finish writing the `render` method so it returns a `div` element that contains an `h1` with the text `Hello React!`.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    // change code below this line



    // change code above this line
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    // change code below this line
    return (
      <div>
        <h1>Hello React!</h1>
      </div>
    );
    // change code above this line
  }
};
```

9. Create a Component with Composition

Description

Now we will look at how we can compose multiple React components together. Imagine you are building an App and have created three components, a `Navbar`, `Dashboard`, and `Footer`. To compose these components together, you could create an `App` *parent* component which renders each of these three components as *children*. To render a component as a child in a React component, you include the component name written as a custom HTML tag in the JSX. For example, in the `render` method you could write:

> return (
> <App>
> <Navbar />
> <Dashboard />
> <Footer />

> </App>
>)

When React encounters a custom HTML tag that references another component (a component name wrapped in `< />` like in this example), it renders the markup for that component in the location of the tag. This should illustrate the parent/child relationship between the `App` component and the `Navbar`, `Dashboard`, and `Footer`.

Instructions

In the code editor, there is a simple functional component called `ChildComponent` and a React component called `ParentComponent`. Compose the two together by rendering the `ChildComponent` within the `ParentComponent`. Make sure to close the `ChildComponent` tag with a forward slash. **Note:** `ChildComponent` is defined with an ES6 arrow function because this is a very common practice when using React. However, know that this is just a function. If you aren't familiar with the arrow function syntax, please refer to the JavaScript section.

Challenge Seed

```
const ChildComponent = () => {
  return (
    <div>
      <p>I am the child</p>
    </div>
  );
};

class ParentComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h1>I am the parent</h1>
        { /* change code below this line */ }


        { /* change code above this line */ }
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<ParentComponent />, document.getElementById('root'))
```

Solution

```
const ChildComponent = () => {
  return (
    <div>
      <p>I am the child</p>
    </div>
  );
};

class ParentComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h1>I am the parent</h1>
        { /* change code below this line */ }
        <ChildComponent />
        { /* change code above this line */ }
      </div>
    );
  }
};
```

10. Use React to Render Nested Components

Description

The last challenge showed a simple way to compose two components, but there are many different ways you can compose components with React. Component composition is one of React's powerful features. When you work with React, it is important to start thinking about your user interface in terms of components like the App example in the last challenge. You break down your UI into its basic building blocks, and those pieces become the components. This helps to separate the code responsible for the UI from the code responsible for handling your application logic. It can greatly simplify the development and maintenance of complex projects.

Instructions

There are two functional components defined in the code editor, called `TypesOfFruit` and `Fruits` . Take the `TypesOfFruit` component and compose it, or *nest* it, within the `Fruits` component. Then take the `Fruits` component and nest it within the `TypesOfFood` component. The result should be a child component, nested within a parent component, which is nested within a parent component of its own!

Challenge Seed

```
const TypesOfFruit = () => {
  return (
    <div>
      <h2>Fruits:</h2>
      <ul>
        <li>Apples</li>
        <li>Blueberries</li>
        <li>Strawberries</li>
        <li>Bananas</li>
      </ul>
    </div>
  );
};

const Fruits = () => {
  return (
    <div>
      { /* change code below this line */ }

      { /* change code above this line */ }
    </div>
  );
};

class TypesOfFood extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <div>
        <h1>Types of Food:</h1>
        { /* change code below this line */ }

        { /* change code above this line */ }
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<TypesOfFood />, document.getElementById('root'))
```

Solution

```
const TypesOfFruit = () => {
  return (
    <div>
      <h2>Fruits:</h2>
      <ul>
        <li>Apples</li>
        <li>Blueberries</li>
        <li>Strawberries</li>
        <li>Bananas</li>
      </ul>
    </div>
  );
};

const Fruits = () => {
  return (
    <div>
      { /* change code below this line */ }
        <TypesOfFruit />
      { /* change code above this line */ }
    </div>
  );
};

class TypesOfFood extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <div>
        <h1>Types of Food:</h1>
        { /* change code below this line */ }
        <Fruits />
        { /* change code above this line */ }
      </div>
    );
  }
};
```

11. Compose React Components

Description

As the challenges continue to use more complex compositions with React components and JSX, there is one important point to note. Rendering ES6 style class components within other components is no different than rendering the simple components you used in the last few challenges. You can render JSX elements, stateless functional components, and ES6 class components within other components.

Instructions

In the code editor, the `TypesOfFood` component is already rendering a component called `Vegetables`. Also, there is the `Fruits` component from the last challenge. Nest two components inside of `Fruits` — first `NonCitrus`, and then `Citrus`. Both of these components are provided for you in the background. Next, nest the `Fruits` class component into the `TypesOfFood` component, below the `h1` header and above `Vegetables`. The result should be a series of nested components, which uses two different component types.

Challenge Seed

```
class Fruits extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h2>Fruits:</h2>
        { /* change code below this line */ }

        { /* change code above this line */ }
      </div>
    );
  }
};

class TypesOfFood extends React.Component {
  constructor(props) {
     super(props);
  }
  render() {
    return (
      <div>
        <h1>Types of Food:</h1>
        { /* change code below this line */ }

        { /* change code above this line */ }
        <Vegetables />
      </div>
    );
  }
};
```

Before Test

```
class NonCitrus extends React.Component {
  render() {
    return (
      <div>
        <h4>Non-Citrus:</h4>
        <ul>
          <li>Apples</li>
          <li>Blueberries</li>
          <li>Strawberries</li>
          <li>Bananas</li>
        </ul>
      </div>
    );
  }
};
class Citrus extends React.Component {
  render() {
    return (
      <div>
        <h4>Citrus:</h4>
        <ul>
          <li>Lemon</li>
          <li>Lime</li>
          <li>Orange</li>
          <li>Grapefruit</li>
        </ul>
      </div>
    );
  }
};
class Vegetables extends React.Component {
  render() {
    return (
      <div>
        <h2>Vegetables:</h2>
        <ul>
```

```
          <li>Brussel Sprouts</li>
          <li>Broccoli</li>
          <li>Squash</li>
        </ul>
      </div>
    );
     }
};
```

After Test

```
ReactDOM.render(<TypesOfFood />, document.getElementById('root'))
```

Solution

```
class Fruits extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h2>Fruits:</h2>
        { /* change code below this line */ }
        <NonCitrus />
        <Citrus />
        { /* change code above this line */ }
      </div>
    )
  }
}

class TypesOfFood extends React.Component {
  constructor(props) {
     super(props);
  }
    render() {
      return (
        <div>
        <h1>Types of Food:</h1>
          { /* change code below this line */ }
          <Fruits />
          { /* change code above this line */ }
          <Vegetables />
        </div>
      );
    }
};
```

12. Render a Class Component to the DOM

Description

You may remember using the ReactDOM API in an earlier challenge to render JSX elements to the DOM. The process for rendering React components will look very similar. The past few challenges focused on components and composition, so the rendering was done for you behind the scenes. However, none of the React code you write will render to the DOM without making a call to the ReactDOM API. Here's a refresher on the syntax: `ReactDOM.render(componentToRender, targetNode)`. The first argument is the React component that you want to render. The second argument is the DOM node that you want to render that component within. React components are passed into `ReactDOM.render()` a little differently than JSX elements. For JSX elements, you pass in the name of the element that you want to render. However, for React components, you need to use the same syntax as if you were rendering a nested component, for example `ReactDOM.render(<ComponentToRender />, targetNode)`. You use this syntax for both ES6 class components and functional components.

Instructions

Both the `Fruits` and `Vegetables` components are defined for you behind the scenes. Render both components as children of the `TypesOfFood` component, then render `TypesOfFood` to the DOM. There is a `div` with `id='challenge-node'` available for you to use.

Challenge Seed

```
class TypesOfFood extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h1>Types of Food:</h1>
        {/* change code below this line */}

        {/* change code above this line */}
      </div>
    );
  }
};

// change code below this line
```

Before Test

```
const Fruits = () => {
  return (
    <div>
      <h2>Fruits:</h2>
      <h4>Non-Citrus:</h4>
        <ul>
          <li>Apples</li>
          <li>Blueberries</li>
          <li>Strawberries</li>
          <li>Bananas</li>
        </ul>
      <h4>Citrus:</h4>
        <ul>
          <li>Lemon</li>
          <li>Lime</li>
          <li>Orange</li>
          <li>Grapefruit</li>
        </ul>
    </div>
  );
};
const Vegetables = () => {
  return (
    <div>
      <h2>Vegetables:</h2>
      <ul>
        <li>Brussel Sprouts</li>
        <li>Broccoli</li>
        <li>Squash</li>
      </ul>
    </div>
  );
};
```

Solution

```
class TypesOfFood extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h1>Types of Food:</h1>
        {/* change code below this line */}
          <Fruits />
           <Vegetables />
         {/* change code above this line */}
      </div>
    );
  }
};

// change code below this line
ReactDOM.render(<TypesOfFood />, document.getElementById('challenge-node'));
```

13. Write a React Component from Scratch

Description

Now that you've learned the basics of JSX and React components, it's time to write a component on your own. React components are the core building blocks of React applications so it's important to become very familiar with writing them. Remember, a typical React component is an ES6 `class` which extends `React.Component`. It has a render method that returns HTML (from JSX) or `null`. This is the basic form of a React component. Once you understand this well, you will be prepared to start building more complex React projects.

Instructions

Define a class `MyComponent` that extends `React.Component`. Its render method should return a `div` that contains an `h1` tag with the text: `My First React Component!` in it. Use this text exactly, the case and punctuation matter. Make sure to call the constructor for your component, too. Render this component to the DOM using `ReactDOM.render()`. There is a `div` with `id='challenge-node'` available for you to use.

Challenge Seed

```
// change code below this line
```

Solution

```
// change code below this line
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h1>My First React Component!</h1>
      </div>
    );
  }
};

ReactDOM.render(<MyComponent />, document.getElementById('challenge-node'));
```

14. Pass Props to a Stateless Functional Component

Description

The previous challenges covered a lot about creating and composing JSX elements, functional components, and ES6 style class components in React. With this foundation, it's time to look at another feature very common in React: **props**. In React, you can pass props, or properties, to child components. Say you have an `App` component which renders a child component called `Welcome` which is a stateless functional component. You can pass `Welcome` a `user` property by writing:

> <App>
> <Welcome user='Mark' />
> </App>

You use **custom HTML attributes** created by you and supported by React to be passed to the component. In this case, the created property `user` is passed to the component `Welcome`. Since `Welcome` is a stateless functional component, it has access to this value like so:

> const Welcome = (props) => <h1>Hello, {props.user}!</h1>

It is standard to call this value `props` and when dealing with stateless functional components, you basically consider it as an argument to a function which returns JSX. You can access the value of the argument in the function body. With class components, you will see this is a little different.

Instructions

There are `Calendar` and `CurrentDate` components in the code editor. When rendering `CurrentDate` from the `Calendar` component, pass in a property of `date` assigned to the current date from JavaScript's `Date` object. Then access this `prop` in the `CurrentDate` component, showing its value within the `p` tags. Note that for `prop` values to be evaluated as JavaScript, they must be enclosed in curly brackets, for instance `date={Date()}`.

Challenge Seed

```
const CurrentDate = (props) => {
  return (
    <div>
      { /* change code below this line */ }
      <p>The current date is: </p>
      { /* change code above this line */ }
    </div>
  );
};

class Calendar extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h3>What date is it?</h3>
        { /* change code below this line */ }
        <CurrentDate />
        { /* change code above this line */ }
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<Calendar />, document.getElementById('root'))
```

Solution

```
const CurrentDate = (props) => {
  return (
    <div>
      { /* change code below this line */ }
      <p>The current date is: {props.date}</p>
      { /* change code above this line */ }
    </div>
  );
};

class Calendar extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h3>What date is it?</h3>
        { /* change code below this line */ }
        <CurrentDate date={Date()} />
        { /* change code above this line */ }
      </div>
    );
  }
};
```

15. Pass an Array as Props

Description

The last challenge demonstrated how to pass information from a parent component to a child component as `props` or properties. This challenge looks at how arrays can be passed as `props` . To pass an array to a JSX element, it must be treated as JavaScript and wrapped in curly braces.

> <ParentComponent>
> <ChildComponent colors={["green", "blue", "red"]} />
> </ParentComponent>

The child component then has access to the array property `colors` . Array methods such as `join()` can be used when accessing the property. `const ChildComponent = (props) => <p>{props.colors.join(', ')}</p>` This will join all `colors` array items into a comma separated string and produce: `<p>green, blue, red</p>` Later, we will learn about other common methods to render arrays of data in React.

Instructions

There are `List` and `ToDo` components in the code editor. When rendering each `List` from the `ToDo` component, pass in a `tasks` property assigned to an array of to-do tasks, for example `["walk dog", "workout"]` . Then access this `tasks` array in the `List` component, showing its value within the `p` element. Use `join(", ")` to display the `props.tasks` array in the `p` element as a comma separated list. Today's list should have at least 2 tasks and tomorrow's should have at least 3 tasks.

Challenge Seed

```
const List= (props) => {
  { /* change code below this line */ }
  return <p>{}</p>
  { /* change code above this line */ }
};

class ToDo extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h1>To Do Lists</h1>
        <h2>Today</h2>
        { /* change code below this line */ }
        <List/>
        <h2>Tomorrow</h2>
        <List/>
        { /* change code above this line */ }
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<ToDo />, document.getElementById('root'))
```

Solution

```
const List= (props) => {
  return <p>{props.tasks.join(', ')}</p>
};

class ToDo extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h1>To Do Lists</h1>
        <h2>Today</h2>
        <List tasks={['study', 'exercise']} />
        <h2>Tomorrow</h2>
        <List tasks={['call Sam', 'grocery shopping', 'order tickets']} />
      </div>
    );
  }
};
```

16. Use Default Props

Description

React also has an option to set default props. You can assign default props to a component as a property on the component itself and React assigns the default prop if necessary. This allows you to specify what a prop value should be if no value is explicitly provided. For example, if you declare `MyComponent.defaultProps = { location: 'San Francisco' }`, you have defined a location prop that's set to the string `San Francisco`, unless you specify otherwise. React assigns default props if props are undefined, but if you pass `null` as the value for a prop, it will remain `null`.

Instructions

The code editor shows a `ShoppingCart` component. Define default props on this component which specify a prop `items` with a value of `0`.

Challenge Seed

```
const ShoppingCart = (props) => {
  return (
    <div>
      <h1>Shopping Cart Component</h1>
    </div>
  )
};
// change code below this line
```

After Test

```
ReactDOM.render(<ShoppingCart />, document.getElementById('root'))
```

Solution

```
const ShoppingCart = (props) => {
  return (
    <div>
      <h1>Shopping Cart Component</h1>
    </div>
  )
};

// change code below this line
ShoppingCart.defaultProps = {
  items: 0
}
```

17. Override Default Props

Description

The ability to set default props is a useful feature in React. The way to override the default props is to explicitly set the prop values for a component.

Instructions

The `ShoppingCart` component now renders a child component `Items` . This `Items` component has a default prop `quantity` set to the integer `0` . Override the default prop by passing in a value of `10` for `quantity` .
Note: Remember that the syntax to add a prop to a component looks similar to how you add HTML attributes. However, since the value for `quantity` is an integer, it won't go in quotes but it should be wrapped in curly braces. For example, `{100}` . This syntax tells JSX to interpret the value within the braces directly as JavaScript.

Challenge Seed

```
const Items = (props) => {
  return <h1>Current Quantity of Items in Cart: {props.quantity}</h1>
}

Items.defaultProps = {
  quantity: 0
}

class ShoppingCart extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    { /* change code below this line */ }
    return <Items />
    { /* change code above this line */ }
  }
};
```

After Test

```
ReactDOM.render(<ShoppingCart />, document.getElementById('root'))
```

Solution

```
const Items = (props) => {
  return <h1>Current Quantity of Items in Cart: {props.quantity}</h1>
}

Items.defaultProps = {
  quantity: 0
}

class ShoppingCart extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    { /* change code below this line */ }
    return <Items quantity = {10} />
    { /* change code above this line */ }
  }
};
```

18. Use PropTypes to Define the Props You Expect

Description

React provides useful type-checking features to verify that components receive props of the correct type. For example, your application makes an API call to retrieve data that you expect to be in an array, which is then passed to a component as a prop. You can set `propTypes` on your component to require the data to be of type `array`. This will throw a useful warning when the data is of any other type. It's considered a best practice to set `propTypes` when you know the type of a prop ahead of time. You can define a `propTypes` property for a component in the same way you defined `defaultProps`. Doing this will check that props of a given key are present with a given type. Here's an example to require the type `function` for a prop called `handleClick`: `MyComponent.propTypes = { handleClick: PropTypes.func.isRequired }` In the example above, the `PropTypes.func` part checks that `handleClick` is a function. Adding `isRequired` tells React that `handleClick` is a required property for that component. You will see a warning if that prop isn't provided. Also notice that `func` represents `function`. Among the seven JavaScript primitive types, `function` and `boolean` (written as `bool`) are the only two that use unusual spelling. In addition to the primitive types, there are other types available. For example, you can check that a prop is a React element. Please refer to the [documentation](https://reactjs.org/docs/jsx-in-depth.html#specifying-the-react-element-type) for all of the options. **Note:** As of React v15.5.0, `PropTypes` is imported independently from React, like this: `import React, { PropTypes } from 'react';`

Instructions

Define `propTypes` for the `Items` component to require `quantity` as a prop and verify that it is of type `number`.

Challenge Seed

```
const Items = (props) => {
  return <h1>Current Quantity of Items in Cart: {props.quantity}</h1>
};

// change code below this line

// change code above this line

Items.defaultProps = {
  quantity: 0
};

class ShoppingCart extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <Items />
  }
};
```

Before Test

```
var PropTypes = {
  number: { isRequired: true }
};
```

After Test

```
ReactDOM.render(<ShoppingCart />, document.getElementById('root'))
```

Solution

```
const Items = (props) => {
  return <h1>Current Quantity of Items in Cart: {props.quantity}</h1>
};

// change code below this line
Items.propTypes = {
  quantity: PropTypes.number.isRequired
};
// change code above this line

Items.defaultProps = {
  quantity: 0
};

class ShoppingCart extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <Items />
  }
};
```

19. Access Props Using this.props

Description

The last several challenges covered the basic ways to pass props to child components. But what if the child component that you're passing a prop to is an ES6 class component, rather than a stateless functional component? The ES6 class component uses a slightly different convention to access props. Anytime you refer to a class component within itself, you use the `this` keyword. To access props within a class component, you preface the code that you use to access it with `this` . For example, if an ES6 class component has a prop called `data` , you write `{this.props.data}` in JSX.

Instructions

Render an instance of the `ReturnTempPassword` component in the parent component `ResetPassword` . Here, give `ReturnTempPassword` a prop of `tempPassword` and assign it a value of a string that is at least 8 characters long. Within the child, `ReturnTempPassword` , access the `tempPassword` prop within the `strong` tags to make sure the user sees the temporary password.

Challenge Seed

```
class ReturnTempPassword extends React.Component {
  constructor(props) {
    super(props);

  }
  render() {
    return (
        <div>
            { /* change code below this line */ }
            <p>Your temporary password is: <strong></strong></p>
            { /* change code above this line */ }
        </div>
    );
  }
};

class ResetPassword extends React.Component {
  constructor(props) {
    super(props);

  }
  render() {
    return (
        <div>
          <h2>Reset Password</h2>
          <h3>We've generated a new temporary password for you.</h3>
          <h3>Please reset this password from your account settings ASAP.</h3>
          { /* change code below this line */ }

          { /* change code above this line */ }
        </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<ResetPassword />, document.getElementById('root'))
```

Solution

```
class ReturnTempPassword extends React.Component {
  constructor(props) {
    super(props);

  }
  render() {
    return (
        <div>
            <p>Your temporary password is: <strong>{this.props.tempPassword}</strong></p>
        </div>
    );
  }
};

class ResetPassword extends React.Component {
  constructor(props) {
    super(props);

  }
  render() {
    return (
        <div>
          <h2>Reset Password</h2>
          <h3>We've generated a new temporary password for you.</h3>
          <h3>Please reset this password from your account settings ASAP.</h3>
          { /* change code below this line */ }
          <ReturnTempPassword tempPassword="serrPbqrPnzc" />
          { /* change code above this line */ }
        </div>
    );
  }
};
```

20. Review Using Props with Stateless Functional Components

Description

Except for the last challenge, you've been passing props to stateless functional components. These components act like pure functions. They accept props as input and return the same view every time they are passed the same props. You may be wondering what state is, and the next challenge will cover it in more detail. Before that, here's a review of the terminology for components. A *stateless functional component* is any function you write which accepts props and returns JSX. A *stateless component*, on the other hand, is a class that extends `React.Component` , but does not use internal state (covered in the next challenge). Finally, a *stateful component* is any component that does maintain its own internal state. You may see stateful components referred to simply as components or React components. A common pattern is to try to minimize statefulness and to create stateless functional components wherever possible. This helps contain your state management to a specific area of your application. In turn, this improves development and maintenance of your app by making it easier to follow how changes to state affect its behavior.

Instructions

The code editor has a `CampSite` component that renders a `Camper` component as a child. Define the `Camper` component and assign it default props of `{ name: 'CamperBot' }` . Inside the `Camper` component, render any code that you want, but make sure to have one `p` element that includes only the `name` value that is passed in as a `prop` . Finally, define `propTypes` on the `Camper` component to require `name` to be provided as a prop and verify that it is of type `string` .

Challenge Seed

```
class CampSite extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <Camper/>
      </div>
    );
  }
};
// change code below this line
```

Before Test

```
var PropTypes = {
   string: { isRequired: true }
};
```

After Test

```
ReactDOM.render(<CampSite />, document.getElementById('root'))
```

Solution

```
class CampSite extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <Camper/>
      </div>
    );
  }
};
// change code below this line

const Camper = (props) => {
   return (
     <div>
       <p>{props.name}</p>
     </div>
   );
};

Camper.propTypes = {
  name: PropTypes.string.isRequired
};

Camper.defaultProps = {
  name: 'CamperBot'
};
```

21. Create a Stateful Component

Description

One of the most important topics in React is `state`. State consists of any data your application needs to know about, that can change over time. You want your apps to respond to state changes and present an updated UI when necessary. React offers a nice solution for the state management of modern web applications. You create state in a React component by declaring a `state` property on the component class in its `constructor`. This initializes the component with `state` when it is created. The `state` property must be set to a JavaScript `object`. Declaring it looks like this:

> this.state = {
>   // describe your state here
> } You have access to the `state` object throughout the life of your component. You can update it, render it in your UI, and pass it as props to child components. The `state` object can be as complex or as simple as you need it to be. Note that you must create a class component by extending `React.Component` in order to create `state` like this.

Instructions

There is a component in the code editor that is trying to render a `name` property from its `state`. However, there is no `state` defined. Initialize the component with `state` in the `constructor` and assign your name to a property of `name`.

Challenge Seed

```
class StatefulComponent extends React.Component {
  constructor(props) {
    super(props);
    // initialize state here

  }
  render() {
    return (
      <div>
        <h1>{this.state.name}</h1>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<StatefulComponent />, document.getElementById('root'))
```

Solution

```
class StatefulComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'freeCodeCamp!'
    }
  }
  render() {
    return (
      <div>
        <h1>{this.state.name}</h1>
      </div>
    );
  }
};
```

22. Render State in the User Interface

Description

Once you define a component's initial state, you can display any part of it in the UI that is rendered. If a component is stateful, it will always have access to the data in `state` in its `render()` method. You can access the data with `this.state`. If you want to access a state value within the `return` of the render method, you have to enclose the value in curly braces. `State` is one of the most powerful features of components in React. It allows you to track important data in your app and render a UI in response to changes in this data. If your data changes, your UI will change. React uses what is called a virtual DOM, to keep track of changes behind the scenes. When state data updates, it triggers a re-render of the components using that data - including child components that received the data as a prop. React updates the actual DOM, but only where necessary. This means you don't have to worry about changing the DOM. You simply declare what the UI should look like. Note that if you make a component stateful, no other components are aware of its `state`. Its `state` is completely encapsulated, or local to that component, unless you pass state data to a child component as `props`. This notion of encapsulated `state` is very important because it allows you to write certain logic, then have that logic contained and isolated in one place in your code.

Instructions

In the code editor, `MyComponent` is already stateful. Define an `h1` tag in the component's render method which renders the value of `name` from the component's state. **Note:** The `h1` should only render the value from `state` and nothing else. In JSX, any code you write with curly braces `{ }` will be treated as JavaScript. So to access the value from `state` just enclose the reference in curly braces.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'freeCodeCamp'
    }
  }
  render() {
    return (
      <div>
        { /* change code below this line */ }

        { /* change code above this line */ }
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'freeCodeCamp'
    }
  }
  render() {
    return (
      <div>
        { /* change code below this line */ }
        <h1>{this.state.name}</h1>
        { /* change code above this line */ }
      </div>
    );
  }
};
```

23. Render State in the User Interface Another Way

Description

There is another way to access `state` in a component. In the `render()` method, before the `return` statement, you can write JavaScript directly. For example, you could declare functions, access data from `state` or `props`, perform computations on this data, and so on. Then, you can assign any data to variables, which you have access to in the `return` statement.

Instructions

In the `MyComponent` render method, define a `const` called `name` and set it equal to the name value in the component's `state`. Because you can write JavaScript directly in this part of the code, you don't have to enclose this reference in curly braces. Next, in the return statement, render this value in an `h1` tag using the variable `name`. Remember, you need to use the JSX syntax (curly braces for JavaScript) in the return statement.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'freeCodeCamp'
    }
  }
  render() {
    // change code below this line

    // change code above this line
```

```
      return (
        <div>
          { /* change code below this line */ }

          { /* change code above this line */ }
        </div>
      );
    }
  };
```

After Test

```
  ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
  class MyComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        name: 'freeCodeCamp'
      }
    }
    render() {
      // change code below this line
      const name = this.state.name;
      // change code above this line
      return (
        <div>
          { /* change code below this line */ }
          <h1>{name}</h1>
          { /* change code above this line */ }
        </div>
      );
    }
  };
```

24. Set State with this.setState

Description

The previous challenges covered component `state` and how to initialize state in the `constructor`. There is also a way to change the component's `state`. React provides a method for updating component `state` called `setState`. You call the `setState` method within your component class like so: `this.setState()`, passing in an object with key-value pairs. The keys are your state properties and the values are the updated state data. For instance, if we were storing a `username` in state and wanted to update it, it would look like this:

> this.setState({
>   username: 'Lewis'
> });

React expects you to never modify `state` directly, instead always use `this.setState()` when state changes occur. Also, you should note that React may batch multiple state updates in order to improve performance. What this means is that state updates through the `setState` method can be asynchronous. There is an alternative syntax for the `setState` method which provides a way around this problem. This is rarely needed but it's good to keep it in mind! Please consult the React documentation for further details.

Instructions

There is a `button` element in the code editor which has an `onClick()` handler. This handler is triggered when the `button` receives a click event in the browser, and runs the `handleClick` method defined on `MyComponent`. Within the `handleClick` method, update the component `state` using `this.setState()`. Set the `name` property in `state` to equal the string `React Rocks!` . Click the button and watch the rendered state update. Don't worry if you don't fully understand how the click handler code works at this point. It's covered in upcoming challenges.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'Initial State'
    };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    // change code below this line

    // change code above this line
  }
  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Click Me</button>
        <h1>{this.state.name}</h1>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'Initial State'
    };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
     // change code below this line
    this.setState({
      name: 'React Rocks!'
    });
    // change code above this line
  }
  render() {
    return (
      <div>
        <button onClick = {this.handleClick}>Click Me</button>
        <h1>{this.state.name}</h1>
      </div>
    );
  }
};
```

25. Bind 'this' to a Class Method

Description

In addition to setting and updating `state`, you can also define methods for your component class. A class method typically needs to use the `this` keyword so it can access properties on the class (such as `state` and `props`) inside the scope of the method. There are a few ways to allow your class methods to access `this`. One common way is to explicitly bind `this` in the constructor so `this` becomes bound to the class methods when the component is initialized. You may have noticed the last challenge used `this.handleClick = this.handleClick.bind(this)` for its `handleClick` method in the constructor. Then, when you call a function like `this.setState()` within your class method, `this` refers to the class and will not be `undefined`. **Note:** The `this` keyword is one of the most confusing aspects of JavaScript but it plays an important role in React. Although its behavior here is totally normal, these lessons aren't the place for an in-depth review of `this` so please refer to other lessons if the above is confusing!

Instructions

The code editor has a component with a `state` that keeps track of an item count. It also has a method which allows you to increment this item count. However, the method doesn't work because it's using the `this` keyword that is undefined. Fix it by explicitly binding `this` to the `addItem()` method in the component's constructor. Next, add a click handler to the `button` element in the render method. It should trigger the `addItem()` method when the button receives a click event. Remember that the method you pass to the `onClick` handler needs curly braces because it should be interpreted directly as JavaScript. Once you complete the above steps you should be able to click the button and see the item count increment in the HTML.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      itemCount: 0
    };
    // change code below this line

    // change code above this line
  }
  addItem() {
    this.setState({
      itemCount: this.state.itemCount + 1
    });
  }
  render() {
    return (
      <div>
        { /* change code below this line */ }
        <button>Click Me</button>
        { /* change code above this line */ }
        <h1>Current Item Count: {this.state.itemCount}</h1>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      itemCount: 0
    };
    this.addItem = this.addItem.bind(this);
  }
  addItem() {
    this.setState({
      itemCount: this.state.itemCount + 1
    });
  }
  render() {
    return (
      <div>
        <button onClick = {this.addItem}>Click Me</button>
        <h1>Current Item Count: {this.state.itemCount}</h1>
      </div>
    );
  }
};
```

26. Use State to Toggle an Element

Description

You can use `state` in React applications in more complex ways than what you've seen so far. One example is to monitor the status of a value, then render the UI conditionally based on this value. There are several different ways to accomplish this, and the code editor shows one method.

Instructions

`MyComponent` has a `visibility` property which is initialized to `false` . The render method returns one view if the value of `visibility` is true, and a different view if it is false. Currently, there is no way of updating the `visibility` property in the component's `state` . The value should toggle back and forth between true and false. There is a click handler on the button which triggers a class method called `toggleVisibility()` . Define this method so the `state` of `visibility` toggles to the opposite value when the method is called. If `visibility` is `false` , the method sets it to `true` , and vice versa. Finally, click the button to see the conditional rendering of the component based on its `state` . **Hint:** Don't forget to bind the `this` keyword to the method in the constructor!

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      visibility: false
    };
    // change code below this line

    // change code above this line
  }
  // change code below this line

  // change code above this line
  render() {
    if (this.state.visibility) {
      return (
        <div>
          <button onClick={this.toggleVisibility}>Click Me</button>
          <h1>Now you see me!</h1>
        </div>
      );
    } else {
      return (
        <div>
          <button onClick={this.toggleVisibility}>Click Me</button>
        </div>
      );
    }
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      visibility: false
    };
    this.toggleVisibility = this.toggleVisibility.bind(this);
   }
  toggleVisibility() {
    this.setState({
      visibility: !this.state.visibility
    });
  }
  render() {
    if (this.state.visibility) {
      return (
        <div>
          <button onClick = {this.toggleVisibility}>Click Me</button>
          <h1>Now you see me!</h1>
        </div>
      );
    } else {
      return (
        <div>
          <button onClick = {this.toggleVisibility}>Click Me</button>
        </div>
      );
    }
  }
};
```

27. Write a Simple Counter

Description

You can design a more complex stateful component by combining the concepts covered so far. These include initializing `state`, writing methods that set `state`, and assigning click handlers to trigger these methods.

Instructions

The `Counter` component keeps track of a `count` value in `state`. There are two buttons which call methods `increment()` and `decrement()`. Write these methods so the counter value is incremented or decremented by 1 when the appropriate button is clicked. Also, create a `reset()` method so when the reset button is clicked, the count is set to 0. **Note:** Make sure you don't modify the `classNames` of the buttons. Also, remember to add the necessary bindings for the newly-created methods in the constructor.

Challenge Seed

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    // change code below this line

    // change code above this line
  }
  // change code below this line
  // change code above this line
  render() {
    return (
      <div>
        <button className='inc' onClick={this.increment}>Increment!</button>
        <button className='dec' onClick={this.decrement}>Decrement!</button>
        <button className='reset' onClick={this.reset}>Reset</button>
        <h1>Current Count: {this.state.count}</h1>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<Counter />, document.getElementById('root'))
```

Solution

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  this.increment = this.increment.bind(this);
 this.decrement = this.decrement.bind(this);
 this.reset = this.reset.bind(this);
 }
  reset() {
    this.setState({
      count: 0
    });
  }
  increment() {
    this.setState({
      count: this.state.count + 1
    });
  }
  decrement() {
    this.setState({
      count: this.state.count - 1
    });
  }
  render() {
    return (
      <div>
        <button className='inc' onClick={this.increment}>Increment!</button>
        <button className='dec' onClick={this.decrement}>Decrement!</button>
        <button className='reset' onClick={this.reset}>Reset</button>
        <h1>Current Count: {this.state.count}</h1>
      </div>
    );
  }
};
```

28. Create a Controlled Input

Description

Your application may have more complex interactions between `state` and the rendered UI. For example, form control elements for text input, such as `input` and `textarea`, maintain their own state in the DOM as the user types. With React, you can move this mutable state into a React component's `state`. The user's input becomes part of the application `state`, so React controls the value of that input field. Typically, if you have React components with input fields the user can type into, it will be a controlled input form.

Instructions

The code editor has the skeleton of a component called `ControlledInput` to create a controlled `input` element. The component's `state` is already initialized with an `input` property that holds an empty string. This value represents the text a user types into the `input` field. First, create a method called `handleChange()` that has a parameter called `event` . When the method is called, it receives an `event` object that contains a string of text from the `input` element. You can access this string with `event.target.value` inside the method. Update the `input` property of the component's `state` with this new string. In the render method, create the `input` element above the `h4` tag. Add a `value` attribute which is equal to the `input` property of the component's `state` . Then add an `onChange()` event handler set to the `handleChange()` method. When you type in the input box, that text is processed by the `handleChange()` method, set as the `input` property in the local `state` , and rendered as the value in the `input` box on the page. The component `state` is the single source of truth regarding the input data. Last but not least, don't forget to add the necessary bindings in the constructor.

Challenge Seed

```
class ControlledInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: ''
    };
    // change code below this line

    // change code above this line
  }
  // change code below this line

  // change code above this line
  render() {
    return (
      <div>
        { /* change code below this line */}

        { /* change code above this line */}
        <h4>Controlled Input:</h4>
        <p>{this.state.input}</p>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<ControlledInput />, document.getElementById('root'))
```

Solution

```
class ControlledInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: ''
    };
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  render() {
    return (
      <div>
        <input
          value={this.state.input}
          onChange={this.handleChange} />
        <h4>Controlled Input:</h4>

        <p>{this.state.input}</p>
      </div>
    );
  }
};
```

29. Create a Controlled Form

Description

The last challenge showed that React can control the internal state for certain elements like `input` and `textarea`, which makes them controlled components. This applies to other form elements as well, including the regular HTML `form` element.

Instructions

The `MyForm` component is set up with an empty `form` with a submit handler. The submit handler will be called when the form is submitted. We've added a button which submits the form. You can see it has the `type` set to `submit` indicating it is the button controlling the form. Add the `input` element in the `form` and set its `value` and `onChange()` attributes like the last challenge. You should then complete the `handleSubmit` method so that it sets the component state property `submit` to the current input value in the local `state`. **Note:** You also must call `event.preventDefault()` in the submit handler, to prevent the default form submit behavior which will refresh the web page. Finally, create an `h1` tag after the `form` which renders the `submit` value from the component's `state`. You can then type in the form and click the button (or press enter), and you should see your input rendered to the page.

Challenge Seed

```
class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      submit: ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  handleSubmit(event) {
    // change code below this line

    // change code above this line
  }
  render() {
    return (
      <div>
        <form onSubmit={this.handleSubmit}>
          { /* change code below this line */ }

          { /* change code above this line */ }
          <button type='submit'>Submit!</button>
        </form>
        { /* change code below this line */ }

        { /* change code above this line */ }
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyForm />, document.getElementById('root'))
```

Solution

```
class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      submit: ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  handleSubmit(event) {
    event.preventDefault()
    this.setState({
      submit: this.state.input
    });
  }
  render() {
    return (
      <div>
        <form onSubmit={this.handleSubmit}>
          <input
            value={this.state.input}
            onChange={this.handleChange} />
          <button type='submit'>Submit!</button>
        </form>
        <h1>{this.state.submit}</h1>
      </div>
    );
  }
};
```

30. Pass State as Props to Child Components

Description

You saw a lot of examples that passed props to child JSX elements and child React components in previous challenges. You may be wondering where those props come from. A common pattern is to have a stateful component containing the `state` important to your app, that then renders child components. You want these components to have access to some pieces of that `state`, which are passed in as props. For example, maybe you have an `App` component that renders a `Navbar`, among other components. In your `App`, you have `state` that contains a lot of user information, but the `Navbar` only needs access to the user's username so it can display it. You pass that piece of `state` to the `Navbar` component as a prop. This pattern illustrates some important paradigms in React. The first is *unidirectional data flow*. State flows in one direction down the tree of your application's components, from the stateful parent component to child components. The child components only receive the state data they need. The second is that complex stateful apps can be broken down into just a few, or maybe a single, stateful component. The rest of your components simply receive state from the parent as props, and render a UI from that state. It begins to create a separation where state management is handled in one part of code and UI rendering in another. This principle of separating state logic from UI logic is one of React's key principles. When it's used correctly, it makes the design of complex, stateful applications much easier to manage.

Instructions

The `MyApp` component is stateful and renders a `Navbar` component as a child. Pass the `name` property in its `state` down to the child component, then show the `name` in the `h1` tag that's part of the `Navbar` render method.

Challenge Seed

```
class MyApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'CamperBot'
    }
  }
  render() {
    return (
       <div>
         <Navbar /* your code here */ />
       </div>
    );
  }
};

class Navbar extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
    <div>
      <h1>Hello, my name is: {/* your code here */} </h1>
    </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyApp />, document.getElementById('root'))
```

Solution

```
class MyApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'CamperBot'
    }
  }
  render() {
    return (
       <div>
         <Navbar name={this.state.name}/>
       </div>
    );
  }
};
class Navbar extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
    <div>
      <h1>Hello, my name is: {this.props.name}</h1>
    </div>
    );
  }
};
```

31. Pass a Callback as Props

Description

You can pass `state` as props to child components, but you're not limited to passing data. You can also pass handler functions or any method that's defined on a React component to a child component. This is how you allow child components to interact with their parent components. You pass methods to a child just like a regular prop. It's assigned a name and you have access to that method name under `this.props` in the child component.

Instructions

There are three components outlined in the code editor. The `MyApp` component is the parent that will render the `GetInput` and `RenderInput` child components. Add the `GetInput` component to the render method in `MyApp` , then pass it a prop called `input` assigned to `inputValue` from `MyApp` 's `state` . Also create a prop called `handleChange` and pass the input handler `handleChange` to it. Next, add `RenderInput` to the render method in `MyApp` , then create a prop called `input` and pass the `inputValue` from `state` to it. Once you are finished you will be able to type in the `input` field in the `GetInput` component, which then calls the handler method in its parent via props. This updates the input in the `state` of the parent, which is passed as props to both children. Observe how the data flows between the components and how the single source of truth remains the `state` of the parent component. Admittedly, this example is a bit contrived, but should serve to illustrate how data and callbacks can be passed between React components.

Challenge Seed

```
class MyApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      inputValue: ''
    }
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    this.setState({
      inputValue: event.target.value
    });
  }
  render() {
    return (
       <div>
        { /* change code below this line */ }

        { /* change code above this line */ }
       </div>
    );
  }
};

class GetInput extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h3>Get Input:</h3>
        <input
          value={this.props.input}
          onChange={this.props.handleChange}/>
      </div>
    );
  }
};

class RenderInput extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h3>Input Render:</h3>
        <p>{this.props.input}</p>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyApp />, document.getElementById('root'))
```

Solution

```
class MyApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      inputValue: ''
    }
  this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    this.setState({
      inputValue: event.target.value
    });
  }
  render() {
    return (
       <div>
         <GetInput
           input={this.state.inputValue}
           handleChange={this.handleChange}/>
         <RenderInput
           input={this.state.inputValue}/>
       </div>
    );
  }
};

class GetInput extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h3>Get Input:</h3>
        <input
          value={this.props.input}
          onChange={this.props.handleChange}/>
      </div>
    );
  }
};

class RenderInput extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h3>Input Render:</h3>
        <p>{this.props.input}</p>
      </div>
    );
    }
  };
```

32. Use the Lifecycle Method componentWillMount

Description

React components have several special methods that provide opportunities to perform actions at specific points in the lifecycle of a component. These are called lifecycle methods, or lifecycle hooks, and allow you to catch components at certain points in time. This can be before they are rendered, before they update, before they receive props, before they unmount, and so on. Here is a list of some of the main lifecycle methods: `componentWillMount()` `componentDidMount()` `componentWillReceiveProps()` `shouldComponentUpdate()` `componentWillUpdate()` `componentDidUpdate()` `componentWillUnmount()` The next several lessons will cover some of the basic use cases for these lifecycle methods.

Note: The `componentWillMount` Lifecycle method will be deprecated in a future version of 16.X and removed in version 17. (Source)

Instructions

The `componentWillMount()` method is called before the `render()` method when a component is being mounted to the DOM. Log something to the console within `componentWillMount()` - you may want to have your browser console open to see the output.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  componentWillMount() {
    // change code below this line

    // change code above this line
  }
  render() {
    return <div />
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  componentWillMount() {
    // change code below this line
    console.log('Component is mounting...');
    // change code above this line
  }
  render() {
    return <div />
  }
};
```

33. Use the Lifecycle Method componentDidMount

Description

Most web developers, at some point, need to call an API endpoint to retrieve data. If you're working with React, it's important to know where to perform this action. The best practice with React is to place API calls or any calls to your server in the lifecycle method `componentDidMount()` . This method is called after a component is mounted to the DOM. Any calls to `setState()` here will trigger a re-rendering of your component. When you call an API in this method, and set your state with the data that the API returns, it will automatically trigger an update once you receive the data.

Instructions

There is a mock API call in `componentDidMount()` . It sets state after 2.5 seconds to simulate calling a server to retrieve data. This example requests the current total active users for a site. In the render method, render the value of `activeUsers` in the `h1` . Watch what happens in the preview, and feel free to change the timeout to see the different effects.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activeUsers: null
    };
  }
  componentDidMount() {
    setTimeout( () => {
      this.setState({
        activeUsers: 1273
      });
    }, 2500);
  }
  render() {
    return (
      <div>
        <h1>Active Users: { /* change code here */ }</h1>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activeUsers: null
    };
  }
  componentDidMount() {
    setTimeout( () => {
      this.setState({
        activeUsers: 1273
      });
    }, 2500);
  }
  render() {
    return (
      <div>
        <h1>Active Users: {this.state.activeUsers}</h1>
      </div>
    );
  }
};
```

34. Add Event Listeners

Description

The `componentDidMount()` method is also the best place to attach any event listeners you need to add for specific functionality. React provides a synthetic event system which wraps the native event system present in browsers. This means that the synthetic event system behaves exactly the same regardless of the user's browser - even if the native events may behave differently between different browsers. You've already been using some of these synthetic event handlers such as `onClick()` . React's synthetic event system is great to use for most interactions you'll manage on DOM elements. However, if you want to attach an event handler to the document or window objects, you have to do this directly.

Instructions

Attach an event listener in the `componentDidMount()` method for `keydown` events and have these events trigger the callback `handleKeyPress()` . You can use `document.addEventListener()` which takes the event (in quotes) as the first argument and the callback as the second argument. Then, in `componentWillUnmount()` , remove this same event listener. You can pass the same arguments to `document.removeEventListener()` . It's good practice to use this lifecycle method to do any clean up on React components before they are unmounted and destroyed. Removing event listeners is an example of one such clean up action.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      message: ''
    };
    this.handleEnter = this.handleEnter.bind(this);
    this.handleKeyPress = this.handleKeyPress.bind(this);
  }
  // change code below this line
  componentDidMount() {

  }
  componentWillUnmount() {

  }
  // change code above this line
  handleEnter() {
    this.setState({
      message: this.state.message + 'You pressed the enter key! '
    });
  }
  handleKeyPress(event) {
    if (event.keyCode === 13) {
      this.handleEnter();
    }
  }
  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      message: ''
    };
    this.handleKeyPress = this.handleKeyPress.bind(this);
    this.handleEnter = this.handleEnter.bind(this);  }
  componentDidMount() {
    // change code below this line
    document.addEventListener('keydown', this.handleKeyPress);
    // change code above this line
  }
  componentWillUnmount() {
    // change code below this line
    document.removeEventListener('keydown', this.handleKeyPress);
    // change code above this line
  }
  handleEnter() {
    this.setState({
      message: this.state.message + 'You pressed the enter key! '
    });
  }
  handleKeyPress(event) {
    if (event.keyCode === 13) {
      this.handleEnter();
    }
  }
  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
      </div>
    );
  }
};
```

35. Manage Updates with Lifecycle Methods

Description

Another lifecycle method is `componentWillReceiveProps()` which is called whenever a component is receiving new props. This method receives the new props as an argument, which is usually written as `nextProps` . You can use this argument and compare with `this.props` and perform actions before the component updates. For example, you may call `setState()` locally before the update is processed. Another method is `componentDidUpdate()` , and is called immediately after a component re-renders. Note that rendering and mounting are considered different things in the component lifecycle. When a page first loads, all components are mounted and this is where methods like `componentWillMount()` and `componentDidMount()` are called. After this, as state changes, components re-render themselves. The next challenge covers this in more detail.

Instructions

The child component `Dialog` receives `message` props from its parent, the `Controller` component. Write the `componentWillReceiveProps()` method in the `Dialog` component and have it log `this.props` and `nextProps` to the console. You'll need to pass `nextProps` as an argument to this method and although it's possible to name it anything, name it `nextProps` here. Next, add `componentDidUpdate()` in the `Dialog` component, and log a statement that says the component has updated. This method works similar to `componentWillUpdate()`, which is provided for you. Now click the button to change the message and watch your browser console. The order of the console statements show the order the methods are called. **Note:** You'll need to write the lifecycle methods as normal functions and not as arrow functions to pass the tests (there is also no advantage to writing lifecycle methods as arrow functions).

Challenge Seed

```
class Dialog extends React.Component {
  constructor(props) {
    super(props);
  }
  componentWillUpdate() {
    console.log('Component is about to update...');
  }
  // change code below this line

  // change code above this line
  render() {
    return <h1>{this.props.message}</h1>
  }
};

class Controller extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      message: 'First Message'
    };
    this.changeMessage = this.changeMessage.bind(this);
  }
  changeMessage() {
    this.setState({
      message: 'Second Message'
    });
  }
  render() {
    return (
      <div>
        <button onClick={this.changeMessage}>Update</button>
        <Dialog message={this.state.message}/>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<Controller />, document.getElementById('root'))
```

Solution

```
class Dialog extends React.Component {
  constructor(props) {
    super(props);
  }
  componentWillUpdate() {
    console.log('Component is about to update...');
  }
  // change code below this line
  componentWillReceiveProps(nextProps) {
    console.log(this.props, nextProps);
  }
  componentDidUpdate() {
    console.log('Component re-rendered');
  }
  // change code above this line
  render() {
    return <h1>{this.props.message}</h1>
  }
};

class Controller extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      message: 'First Message'
    };
 this.changeMessage = this.changeMessage.bind(this);
  }
  changeMessage() {
    this.setState({
      message: 'Second Message'
    });
  }
  render() {
    return (
      <div>
        <button onClick={this.changeMessage}>Update</button>
        <Dialog message={this.state.message}/>
      </div>
    );
  }
};
```

36. Optimize Re-Renders with shouldComponentUpdate

Description

So far, if any component receives new `state` or new `props` , it re-renders itself and all its children. This is usually okay. But React provides a lifecycle method you can call when child components receive new `state` or `props` , and declare specifically if the components should update or not. The method is `shouldComponentUpdate()` , and it takes `nextProps` and `nextState` as parameters. This method is a useful way to optimize performance. For example, the default behavior is that your component re-renders when it receives new `props` , even if the `props` haven't changed. You can use `shouldComponentUpdate()` to prevent this by comparing the `props` . The method must return a `boolean` value that tells React whether or not to update the component. You can compare the current props (`this.props`) to the next props (`nextProps`) to determine if you need to update or not, and return `true` or `false` accordingly.

Instructions

The `shouldComponentUpdate()` method is added in a component called `OnlyEvens` . Currently, this method returns `true` so `OnlyEvens` re-renders every time it receives new `props` . Modify the method so `OnlyEvens` updates only if the `value` of its new props is even. Click the `Add` button and watch the order of events in your browser's console as the other lifecycle hooks are triggered.

Challenge Seed

```
class OnlyEvens extends React.Component {
  constructor(props) {
    super(props);
  }
  shouldComponentUpdate(nextProps, nextState) {
    console.log('Should I update?');
     // change code below this line
    return true;
     // change code above this line
  }
  componentWillReceiveProps(nextProps) {
    console.log('Receiving new props...');
  }
  componentDidUpdate() {
    console.log('Component re-rendered.');
  }
  render() {
    return <h1>{this.props.value}</h1>
  }
};

class Controller extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 0
    };
    this.addValue = this.addValue.bind(this);
  }
  addValue() {
    this.setState({
      value: this.state.value + 1
    });
  }
  render() {
    return (
      <div>
        <button onClick={this.addValue}>Add</button>
        <OnlyEvens value={this.state.value}/>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<Controller />, document.getElementById('root'))
```

Solution

```
class OnlyEvens extends React.Component {
  constructor(props) {
    super(props);
  }
  shouldComponentUpdate(nextProps, nextState) {
    console.log('Should I update?');
    // change code below this line
    return nextProps.value % 2 === 0;
    // change code above this line
  }
  componentWillReceiveProps(nextProps) {
    console.log('Receiving new props...');
  }
  componentDidUpdate() {
    console.log('Component re-rendered.');
  }
  render() {
    return <h1>{this.props.value}</h1>
  }
};

class Controller extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 0
    };
  this.addValue = this.addValue.bind(this);
  }
  addValue() {
    this.setState({
      value: this.state.value + 1
    });
  }
  render() {
    return (
      <div>
        <button onClick={this.addValue}>Add</button>
        <OnlyEvens value={this.state.value}/>
      </div>
    );
  }
};
```

37. Introducing Inline Styles

Description

There are other complex concepts that add powerful capabilities to your React code. But you may be wondering about the more simple problem of how to style those JSX elements you create in React. You likely know that it won't be exactly the same as working with HTML because of the way you apply classes to JSX elements. If you import styles from a stylesheet, it isn't much different at all. You apply a class to your JSX element using the `className` attribute, and apply styles to the class in your stylesheet. Another option is to apply ***inline*** styles, which are very common in ReactJS development. You apply inline styles to JSX elements similar to how you do it in HTML, but with a few JSX differences. Here's an example of an inline style in HTML: `<div style="color: yellow; font-size: 16px">Mellow Yellow</div>` JSX elements use the `style` attribute, but because of the way JSX is transpiled, you can't set the value to a `string` . Instead, you set it equal to a JavaScript `object` . Here's an example: `<div style={{color: "yellow", fontSize: 16}}>Mellow Yellow</div>` Notice how we camelCase the "fontSize" property? This is because React will not accept kebab-case keys in the style object. React will apply the correct property name for us in the HTML.

Instructions

Add a `style` attribute to the `div` in the code editor to give the text a color of red and font size of 72px. Note that you can optionally set the font size to be a number, omitting the units "px", or write it as "72px".

Challenge Seed

```
class Colorful extends React.Component {
  render() {
    return (
      <div>Big Red</div>
    );
  }
};
```

After Test

```
ReactDOM.render(<Colorful />, document.getElementById('root'))
```

Solution

```
class Colorful extends React.Component {
  render() {
    return (
      <div style={{color: "red", fontSize: 72}}>Big Red</div>
    );
  }
};
```

38. Add Inline Styles in React

Description

You may have noticed in the last challenge that there were several other syntax differences from HTML inline styles in addition to the `style` attribute set to a JavaScript object. First, the names of certain CSS style properties use camel case. For example, the last challenge set the size of the font with `fontSize` instead of `font-size`. Hyphenated words like `font-size` are invalid syntax for JavaScript object properties, so React uses camel case. As a rule, any hyphenated style properties are written using camel case in JSX. All property value length units (like `height`, `width`, and `fontSize`) are assumed to be in `px` unless otherwise specified. If you want to use `em`, for example, you wrap the value and the units in quotes, like `{fontSize: "4em"}`. Other than the length values that default to `px`, all other property values should be wrapped in quotes.

Instructions

If you have a large set of styles, you can assign a style `object` to a constant to keep your code organized. Uncomment the `styles` constant and declare an `object` with three style properties and their values. Give the `div` a color of `"purple"`, a font-size of `40`, and a border of `"2px solid purple"`. Then set the `style` attribute equal to the `styles` constant.

Challenge Seed

```
// const styles =
// change code above this line
class Colorful extends React.Component {
  render() {
    // change code below this line
    return (
      <div style={{color: "yellow", fontSize: 24}}>Style Me!</div>
    );
    // change code above this line
  }
};
```

After Test

```
ReactDOM.render(<Colorful />, document.getElementById('root'))
```

Solution

```
const styles = {
  color: "purple",
  fontSize: 40,
  border: "2px solid purple"
};
// change code above this line
class Colorful extends React.Component {
  render() {
    // change code below this line
    return (
      <div style={styles}>Style Me!</div>
  // change code above this line
    );
  }
};
```

39. Use Advanced JavaScript in React Render Method

Description

In previous challenges, you learned how to inject JavaScript code into JSX code using curly braces, `{ }`, for tasks like accessing props, passing props, accessing state, inserting comments into your code, and most recently, styling your components. These are all common use cases to put JavaScript in JSX, but they aren't the only way that you can utilize JavaScript code in your React components. You can also write JavaScript directly in your `render` methods, before the `return` statement, ***without*** inserting it inside of curly braces. This is because it is not yet within the JSX code. When you want to use a variable later in the JSX code *inside* the `return` statement, you place the variable name inside curly braces.

Instructions

In the code provided, the `render` method has an array that contains 20 phrases to represent the answers found in the classic 1980's Magic Eight Ball toy. The button click event is bound to the `ask` method, so each time the button is clicked a random number will be generated and stored as the `randomIndex` in state. On line 52, delete the string `"change me!"` and reassign the `answer` const so your code randomly accesses a different index of the `possibleAnswers` array each time the component updates. Finally, insert the `answer` const inside the `p` tags.

Challenge Seed

```
const inputStyle = {
  width: 235,
  margin: 5
}

class MagicEightBall extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      userInput: '',
      randomIndex: ''
    }
    this.ask = this.ask.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }
  ask() {
    if (this.state.userInput) {
      this.setState({
        randomIndex: Math.floor(Math.random() * 20),
        userInput: ''
      });
    }
  }
  handleChange(event) {
    this.setState({
      userInput: event.target.value
    });
  }
  render() {
    const possibleAnswers = [
      'It is certain',
      'It is decidedly so',
      'Without a doubt',
      'Yes, definitely',
      'You may rely on it',
      'As I see it, yes',
      'Outlook good',
      'Yes',
      'Signs point to yes',
      'Reply hazy try again',
      'Ask again later',
      'Better not tell you now',
      'Cannot predict now',
      'Concentrate and ask again',
      'Don\'t count on it',
      'My reply is no',
      'My sources say no',
      'Most likely',
      'Outlook not so good',
      'Very doubtful'
    ];
    const answer = 'change me!' // << change code here
    return (
      <div>
        <input
          type="text"
          value={this.state.userInput}
          onChange={this.handleChange}
          style={inputStyle} /><br />
        <button onClick={this.ask}>
          Ask the Magic Eight Ball!
        </button><br />
        <h3>Answer:</h3>
        <p>
          { /* change code below this line */ }

          { /* change code above this line */ }
        </p>
      </div>
    );
  }
};
```

After Test

```
var possibleAnswers = [ 'It is certain', 'It is decidedly so', 'Without a doubt', 'Yes, definitely',
'You may rely on it', 'As I see it, yes', 'Outlook good', 'Yes', 'Signs point to yes', 'Reply hazy try
again', 'Ask again later', 'Better not tell you now', 'Cannot predict now', 'Concentrate and ask again',
'Don\'t count on it', 'My reply is no', 'My sources say no', 'Outlook not so good','Very doubtful',
'Most likely' ];
ReactDOM.render(<MagicEightBall />, document.getElementById('root'))
```

Solution

```
const inputStyle = {
  width: 235,
  margin: 5
}

class MagicEightBall extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      userInput: '',
      randomIndex: ''
    }
    this.ask = this.ask.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }
  ask() {
    if (this.state.userInput) {
      this.setState({
        randomIndex: Math.floor(Math.random() * 20),
        userInput: ''
      });
    }
  }
  handleChange(event) {
    this.setState({
      userInput: event.target.value
    });
  }
  render() {
    const possibleAnswers = [
      "It is certain", "It is decidedly so", "Without a doubt",
      "Yes, definitely", "You may rely on it", "As I see it, yes",
      "Outlook good", "Yes", "Signs point to yes", "Reply hazy try again",
      "Ask again later", "Better not tell you now", "Cannot predict now",
      "Concentrate and ask again", "Don't count on it", "My reply is no",
      "My sources say no", "Outlook not so good","Very doubtful", "Most likely"
    ];
    const answer = possibleAnswers[this.state.randomIndex];
    return (
      <div>
        <input
          type="text"
          value={this.state.userInput}
          onChange={this.handleChange}
          style={inputStyle} /><br />
        <button onClick={this.ask}>Ask the Magic Eight Ball!</button><br />
        <h3>Answer:</h3>
        <p>
          {answer}
        </p>
      </div>
    );
  }
};
```

40. Render with an If/Else Condition

Description

Another application of using JavaScript to control your rendered view is to tie the elements that are rendered to a condition. When the condition is true, one view renders. When it's false, it's a different view. You can do this with a standard `if/else` statement in the `render()` method of a React component.

Instructions

MyComponent contains a `boolean` in its state which tracks whether you want to display some element in the UI or not. The `button` toggles the state of this value. Currently, it renders the same UI every time. Rewrite the `render()` method with an `if/else` statement so that if `display` is `true` , you return the current markup. Otherwise, return the markup without the `h1` element. **Note:** You must write an `if/else` to pass the tests. Use of the ternary operator will not pass here.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      display: true
    }
    this.toggleDisplay = this.toggleDisplay.bind(this);
  }
  toggleDisplay() {
    this.setState({
      display: !this.state.display
    });
  }
  render() {
    // change code below this line

    return (
       <div>
         <button onClick={this.toggleDisplay}>Toggle Display</button>
         <h1>Displayed!</h1>
       </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      display: true
    }
    this.toggleDisplay = this.toggleDisplay.bind(this);
  }
  toggleDisplay() {
    this.setState({
      display: !this.state.display
    });
  }
  render() {
    // change code below this line
    if (this.state.display) {
      return (
         <div>
           <button onClick={this.toggleDisplay}>Toggle Display</button>
           <h1>Displayed!</h1>
         </div>
      );
    } else {
      return (
        <div>
           <button onClick={this.toggleDisplay}>Toggle Display</button>
         </div>
      );
    }
  }
};
```

41. Use && for a More Concise Conditional

Description

The if/else statements worked in the last challenge, but there's a more concise way to achieve the same result. Imagine that you are tracking several conditions in a component and you want different elements to render depending on each of these conditions. If you write a lot of `else if` statements to return slightly different UIs, you may repeat code which leaves room for error. Instead, you can use the `&&` logical operator to perform conditional logic in a more concise way. This is possible because you want to check if a condition is `true` , and if it is, return some markup. Here's an example: `{condition && <p>markup</p>}` If the `condition` is `true` , the markup will be returned. If the condition is `false` , the operation will immediately return `false` after evaluating the `condition` and return nothing. You can include these statements directly in your JSX and string multiple conditions together by writing `&&` after each one. This allows you to handle more complex conditional logic in your `render()` method without repeating a lot of code.

Instructions

Solve the previous example again, so the `h1` only renders if `display` is `true` , but use the `&&` logical operator instead of an `if/else` statement.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      display: true
    }
    this.toggleDisplay = this.toggleDisplay.bind(this);
  }
  toggleDisplay() {
    this.setState({
      display: !this.state.display
    });
  }
  render() {
    // change code below this line
    return (
       <div>
```

```
          <button onClick={this.toggleDisplay}>Toggle Display</button>
          <h1>Displayed!</h1>
        </div>
      );
    }
  };
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      display: true
    }
 this.toggleDisplay = this.toggleDisplay.bind(this);
  }
  toggleDisplay() {
    this.setState({
      display: !this.state.display
    });
  }
  render() {
    // change code below this line
    return (
       <div>
         <button onClick={this.toggleDisplay}>Toggle Display</button>
         {this.state.display && <h1>Displayed!</h1>}
       </div>
    );
  }
};
```

42. Use a Ternary Expression for Conditional Rendering

Description

Before moving on to dynamic rendering techniques, there's one last way to use built-in JavaScript conditionals to render what you want: the ***ternary operator***. The ternary operator is often utilized as a shortcut for `if/else` statements in JavaScript. They're not quite as robust as traditional `if/else` statements, but they are very popular among React developers. One reason for this is because of how JSX is compiled, `if/else` statements can't be inserted directly into JSX code. You might have noticed this a couple challenges ago — when an `if/else` statement was required, it was always *outside* the `return` statement. Ternary expressions can be an excellent alternative if you want to implement conditional logic within your JSX. Recall that a ternary operator has three parts, but you can combine several ternary expressions together. Here's the basic syntax:

> condition ? expressionIfTrue : expressionIfFalse

Instructions

The code editor has three constants defined within the `CheckUserAge` component's `render()` method. They are called `buttonOne` , `buttonTwo` , and `buttonThree` . Each of these is assigned a simple JSX expression representing a button element. First, initialize the state of `CheckUserAge` with `input` and `userAge` both set to values of an empty string. Once the component is rendering information to the page, users should have a way to interact with it. Within the component's `return` statement, set up a ternary expression that implements the following logic: when the page first loads, render the submit button, `buttonOne` , to the page. Then, when a user enters their age and clicks the button, render a different button based on the age. If a user enters a number less than `18` , render `buttonThree` . If a user enters a number greater than or equal to `18` , render `buttonTwo` .

Challenge Seed

```
const inputStyle = {
  width: 235,
  margin: 5
}

class CheckUserAge extends React.Component {
  constructor(props) {
    super(props);
    // change code below this line

    // change code above this line
    this.submit = this.submit.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(e) {
    this.setState({
      input: e.target.value,
      userAge: ''
    });
  }
  submit() {
    this.setState({
      userAge: this.state.input
    });
  }
  render() {
    const buttonOne = <button onClick={this.submit}>Submit</button>;
    const buttonTwo = <button>You May Enter</button>;
    const buttonThree = <button>You Shall Not Pass</button>;
    return (
      <div>
        <h3>Enter Your Age to Continue</h3>
        <input
          style={inputStyle}
          type="number"
          value={this.state.input}
          onChange={this.handleChange} /><br />
        {
          /* change code here */
        }
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<CheckUserAge />, document.getElementById('root'))
```

Solution

```
const inputStyle = {
  width: 235,
  margin: 5
}

class CheckUserAge extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      userAge: '',
      input: ''
    }
    this.submit = this.submit.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(e) {
    this.setState({
      input: e.target.value,
      userAge: ''
    });
  }
  submit() {
    this.setState({
      userAge: this.state.input
    });
  }
  render() {
    const buttonOne = <button onClick={this.submit}>Submit</button>;
    const buttonTwo = <button>You May Enter</button>;
    const buttonThree = <button>You Shall Not Pass</button>;
    return (
      <div>
        <h3>Enter Your Age to Continue</h3>
        <input
          style={inputStyle}
          type="number"
          value={this.state.input}
          onChange={this.handleChange} /><br />
          {
            this.state.userAge === '' ?
            buttonOne :
            this.state.userAge >= 18 ?
            buttonTwo :
            buttonThree
          }
      </div>
    );
  }
};
```

43. Render Conditionally from Props

Description

So far, you've seen how to use `if/else`, `&&,` `null` and the ternary operator (`condition ? expressionIfTrue : expressionIfFalse`) to make conditional decisions about what to render and when. However, there's one important topic left to discuss that lets you combine any or all of these concepts with another powerful React feature: props. Using props to conditionally render code is very common with React developers — that is, they use the value of a given prop to automatically make decisions about what to render. In this challenge, you'll set up a child component to make rendering decisions based on props. You'll also use the ternary operator, but you can see how several of the other concepts that were covered in the last few challenges might be just as useful in this context.

Instructions

The code editor has two components that are partially defined for you: a parent called `GameOfChance`, and a child called `Results`. They are used to create a simple game where the user presses a button to see if they win or lose. First, you'll need a simple expression that randomly returns a different value every time it is run. You can use `Math.random()`. This method returns a value between `0` (inclusive) and `1` (exclusive) each time it is called. So for 50/50 odds, use `Math.random() > .5` in your expression. Statistically speaking, this expression will return `true` 50% of the time, and `false` the other 50%. On line 30, replace the comment with this expression to complete the variable declaration. Now you have an expression that you can use to make a randomized decision in the code. Next you need to implement this. Render the `Results` component as a child of `GameOfChance`, and pass in `expression` as a prop called `fiftyFifty`. In the `Results` component, write a ternary expression to render the text `"You Win!"` or `"You Lose!"` based on the `fiftyFifty` prop that's being passed in from `GameOfChance`. Finally, make sure the `handleClick()` method is correctly counting each turn so the user knows how many times they've played. This also serves to let the user know the component has actually updated in case they win or lose twice in a row.

Challenge Seed

```
class Results extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <h1>
      {
        /* change code here */
      }
      </h1>
    )
  };
};

class GameOfChance extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      counter: 1
    }
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState({
      counter: 0 // change code here
    });
  }
  render() {
    let expression = null; // change code here
    return (
      <div>
        <button onClick={this.handleClick}>Play Again</button>
        { /* change code below this line */ }

        { /* change code above this line */ }
        <p>{'Turn: ' + this.state.counter}</p>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<GameOfChance />, document.getElementById('root'))
```

Solution

```
class Results extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <h1>
      {
        this.props.fiftyFifty ?
        'You Win!' :
        'You Lose!'
      }
      </h1>
    )
  };
};

class GameOfChance extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      counter: 1
    }
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState({
      counter: this.state.counter + 1
    });
  }
  render() {
    const expression = Math.random() > .5;
    return (
      <div>
        <button onClick={this.handleClick}>Play Again</button>
        <Results fiftyFifty={expression} />
        <p>{'Turn: ' + this.state.counter}</p>
      </div>
    );
  }
};
```

44. Change Inline CSS Conditionally Based on Component State

Description

At this point, you've seen several applications of conditional rendering and the use of inline styles. Here's one more example that combines both of these topics. You can also render CSS conditionally based on the state of a React component. To do this, you check for a condition, and if that condition is met, you modify the styles object that's assigned to the JSX elements in the render method. This paradigm is important to understand because it is a dramatic shift from the more traditional approach of applying styles by modifying DOM elements directly (which is very common with jQuery, for example). In that approach, you must keep track of when elements change and also handle the actual manipulation directly. It can become difficult to keep track of changes, potentially making your UI unpredictable. When you set a style object based on a condition, you describe how the UI should look as a function of the application's state. There is a clear flow of information that only moves in one direction. This is the preferred method when writing applications with React.

Instructions

The code editor has a simple controlled input component with a styled border. You want to style this border red if the user types more than 15 characters of text in the input box. Add a condition to check for this and, if the condition is valid, set the input border style to `3px solid red`. You can try it out by entering text in the input.

Challenge Seed

```
class GateKeeper extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: ''
    };
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    this.setState({ input: event.target.value })
  }
  render() {
    let inputStyle = {
      border: '1px solid black'
    };
    // change code below this line

    // change code above this line
    return (
      <div>
        <h3>Don't Type Too Much:</h3>
        <input
          type="text"
          style={inputStyle}
          value={this.state.input}
          onChange={this.handleChange} />
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<GateKeeper />, document.getElementById('root'))
```

Solution

```
class GateKeeper extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: ''
    };
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    this.setState({ input: event.target.value })
  }
  render() {
    let inputStyle = {
      border: '1px solid black'
    };
    if (this.state.input.length > 15) {
      inputStyle.border = '3px solid red';
    };
    return (
      <div>
        <h3>Don't Type Too Much:</h3>
        <input
          type="text"
          style={inputStyle}
          value={this.state.input}
          onChange={this.handleChange} />
      </div>
    );
  }
};
```

45. Use Array.map() to Dynamically Render Elements

Description

Conditional rendering is useful, but you may need your components to render an unknown number of elements. Often in reactive programming, a programmer has no way to know what the state of an application is until runtime, because so much depends on a user's interaction with that program. Programmers need to write their code to correctly handle that unknown state ahead of time. Using `Array.map()` in React illustrates this concept. For example, you create a simple "To Do List" app. As the programmer, you have no way of knowing how many items a user might have on their list. You need to set up your component to ***dynamically render*** the correct number of list elements long before someone using the program decides that today is laundry day.

Instructions

The code editor has most of the `MyToDoList` component set up. Some of this code should look familiar if you completed the controlled form challenge. You'll notice a `textarea` and a `button`, along with a couple of methods that track their states, but nothing is rendered to the page yet. Inside the `constructor`, create a `this.state` object and define two states: `userInput` should be initialized as an empty string, and `toDoList` should be initialized as an empty array. Next, delete the comment in the `render()` method next to the `items` variable. In its place, map over the `toDoList` array stored in the component's internal state and dynamically render a `li` for each item. Try entering the string `eat, code, sleep, repeat` into the `textarea`, then click the button and see what happens. **Note:** You may know that all sibling child elements created by a mapping operation like this do need to be supplied with a unique `key` attribute. Don't worry, this is the topic of the next challenge.

Challenge Seed

```
const textAreaStyles = {
  width: 235,
  margin: 5
};

class MyToDoList extends React.Component {
  constructor(props) {
    super(props);
    // change code below this line

    // change code above this line
    this.handleSubmit = this.handleSubmit.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }
  handleSubmit() {
    const itemsArray = this.state.userInput.split(',');
    this.setState({
      toDoList: itemsArray
    });
  }
  handleChange(e) {
    this.setState({
      userInput: e.target.value
    });
  }
  render() {
    const items = null; // change code here
    return (
      <div>
        <textarea
          onChange={this.handleChange}
          value={this.state.userInput}
          style={textAreaStyles}
          placeholder="Separate Items With Commas" /><br />
        <button onClick={this.handleSubmit}>Create List</button>
        <h1>My "To Do" List:</h1>
        <ul>
          {items}
        </ul>
      </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyToDoList />, document.getElementById('root'))
```

Solution

```
const textAreaStyles = {
  width: 235,
  margin: 5
};

class MyToDoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      toDoList: [],
      userInput: ''
    }
    this.handleSubmit = this.handleSubmit.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }
  handleSubmit() {
    const itemsArray = this.state.userInput.split(',');
    this.setState({
      toDoList: itemsArray
    });
  }
  handleChange(e) {
    this.setState({
      userInput: e.target.value
    });
  }
  render() {
    const items = this.state.toDoList.map( (item, i) => {
      return <li key={i}>{item}</li>
    });
    return (
      <div>
        <textarea
          onChange={this.handleChange}
          value={this.state.userInput}
          style={textAreaStyles}
          placeholder="Separate Items With Commas" /><br />
        <button onClick={this.handleSubmit}>Create List</button>
        <h1>My "To Do" List:</h1>
        <ul>
          {items}
        </ul>
      </div>
    );
  }
};
```

46. Give Sibling Elements a Unique Key Attribute

Description

The last challenge showed how the `map` method is used to dynamically render a number of elements based on user input. However, there was an important piece missing from that example. When you create an array of elements, each one needs a `key` attribute set to a unique value. React uses these keys to keep track of which items are added, changed, or removed. This helps make the re-rendering process more efficient when the list is modified in any way.

Note: Keys only need to be unique between sibling elements, they don't need to be globally unique in your application.

Instructions

The code editor has an array with some front end frameworks and a stateless functional component named `Frameworks()`. `Frameworks()` needs to map the array to an unordered list, much like in the last challenge. Finish writing the `map` callback to return an `li` element for each framework in the `frontEndFrameworks` array. This time, make sure to give each `li` a `key` attribute, set to a unique value. The `li` elements should also contain text from `frontEndFrameworks`. Normally, you want to make the key something that uniquely identifies the element being rendered. As a last resort the array index may be used, but typically you should try to use a unique identification.

Challenge Seed

```
const frontEndFrameworks = [
  'React',
  'Angular',
  'Ember',
  'Knockout',
  'Backbone',
  'Vue'
];

function Frameworks() {
  const renderFrameworks = null; // change code here
  return (
    <div>
      <h1>Popular Front End JavaScript Frameworks</h1>
      <ul>
        {renderFrameworks}
      </ul>
    </div>
  );
};
```

After Test

```
ReactDOM.render(<Frameworks />, document.getElementById('root'))
```

Solution

```
const frontEndFrameworks = [
  'React',
  'Angular',
  'Ember',
  'Knockout',
  'Backbone',
  'Vue'
];

function Frameworks() {
  const renderFrameworks = frontEndFrameworks.map((fw, i) => <li key={i}>{fw}</li>);
  return (
    <div>
      <h1>Popular Front End JavaScript Frameworks</h1>
      <ul>
        {renderFrameworks}
      </ul>
    </div>
  );
};
```

47. Use Array.filter() to Dynamically Filter an Array

Description

The `map` array method is a powerful tool that you will use often when working with React. Another method related to `map` is `filter` , which filters the contents of an array based on a condition, then returns a new array. For example, if you have an array of users that all have a property `online` which can be set to `true` or `false` , you can filter only those users that are online by writing: `let onlineUsers = users.filter(user => user.online);`

Instructions

In the code editor, `MyComponent` 's `state` is initialized with an array of users. Some users are online and some aren't. Filter the array so you see only the users who are online. To do this, first use `filter` to return a new array containing only the users whose `online` property is `true` . Then, in the `renderOnline` variable, map over the filtered array, and return a `li` element for each user that contains the text of their `username` . Be sure to include a unique `key` as well, like in the last challenges.

Challenge Seed

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      users: [
        {
          username: 'Jeff',
          online: true
        },
        {
          username: 'Alan',
          online: false
        },
        {
          username: 'Mary',
          online: true
        },
        {
          username: 'Jim',
          online: false
        },
        {
          username: 'Sara',
          online: true
        },
        {
          username: 'Laura',
          online: true
        }
      ]
    }
  }
  render() {
    const usersOnline = null; // change code here
    const renderOnline = null; // change code here
    return (
       <div>
         <h1>Current Online Users:</h1>
         <ul>
           {renderOnline}
         </ul>
       </div>
    );
  }
};
```

After Test

```
ReactDOM.render(<MyComponent />, document.getElementById('root'))
```

Solution

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      users: [
        {
          username: 'Jeff',
          online: true
        },
        {
          username: 'Alan',
          online: false
        },
        {
          username: 'Mary',
          online: true
        },
        {
          username: 'Jim',
          online: false
        },
        {
          username: 'Sara',
          online: true
        },
        {
          username: 'Laura',
          online: true
        }
      ]
    }
  }
  render() {
    const usersOnline = this.state.users.filter(user => {
      return user.online;
    });
    const renderOnlineUsers = usersOnline.map(user => {
      return (
        <li key={user.username}>{user.username}</li>
      );
    });
    return (
       <div>
         <h1>Current Online Users:</h1>
         <ul>
          {renderOnlineUsers}
        </ul>
       </div>
    );
  }
};
```

48. Render React on the Server with renderToString

Description

So far, you have been rendering React components on the client. Normally, this is what you will always do. However, there are some use cases where it makes sense to render a React component on the server. Since React is a JavaScript view library and you can run JavaScript on the server with Node, this is possible. In fact, React provides a `renderToString()` method you can use for this purpose. There are two key reasons why rendering on the server may be used in a real world app. First, without doing this, your React apps would consist of a relatively empty HTML file and a large bundle of JavaScript when it's initially loaded to the browser. This may not be ideal for search engines that are trying to index the content of your pages so people can find you. If you render the initial HTML markup on the server and send this to the client, the initial page load contains all of the page's markup which can be crawled by search engines. Second, this creates a faster initial page load experience because the rendered HTML is smaller than the JavaScript code of the entire app. React will still be able to recognize your app and manage it after the initial load.

Instructions

The `renderToString()` method is provided on `ReactDOMServer` , which is available here as a global object. The method takes one argument which is a React element. Use this to render `App` to a string.

Challenge Seed

```
class App extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <div/>
  }
};

// change code below this line
```

Before Test

```
var ReactDOMServer = { renderToString(x) { return null; } };
```

After Test

```
ReactDOM.render(<App />, document.getElementById('root'))
```

Solution

```
class App extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <div/>
  }
};

// change code below this line
ReactDOMServer.renderToString(<App/>);
```

Redux

1. Create a Redux Store

Description

Redux is a state management framework that can be used with a number of different web technologies, including React. In Redux, there is a single state object that's responsible for the entire state of your application. This means if you had a React app with ten components, and each component had its own local state, the entire state of your app would be defined by a single state object housed in the Redux `store`. This is the first important principle to understand when learning Redux: the Redux store is the single source of truth when it comes to application state. This also means that any time any piece of your app wants to update state, it **must** do so through the Redux store. The unidirectional data flow makes it easier to track state management in your app.

Instructions

The Redux `store` is an object which holds and manages application `state`. There is a method called `createStore()` on the Redux object, which you use to create the Redux `store`. This method takes a `reducer` function as a required argument. The `reducer` function is covered in a later challenge, and is already defined for you in the code editor. It simply takes `state` as an argument and returns `state`. Declare a `store` variable and assign it to the `createStore()` method, passing in the `reducer` as an argument. **Note:** The code in the editor uses ES6 default argument syntax to initialize this state to hold a value of `5`. If you're not familiar with default arguments, you can refer to the ES6 section in the Curriculum which covers this topic.

Challenge Seed

```
const reducer = (state = 5) => {
  return state;
}

// Redux methods are available from a Redux object
// For example: Redux.createStore()
// Define the store here:
```

Solution

```
const reducer = (state = 5) => {
  return state;
}

// Redux methods are available from a Redux object
// For example: Redux.createStore()
// Define the store here:

const store = Redux.createStore(reducer);
```

2. Get State from the Redux Store

Description

The Redux store object provides several methods that allow you to interact with it. For example, you can retrieve the current `state` held in the Redux store object with the `getState()` method.

Instructions

The code from the previous challenge is re-written more concisely in the code editor. Use `store.getState()` to retrieve the `state` from the `store`, and assign this to a new variable `currentState`.

Challenge Seed

```
const store = Redux.createStore(
  (state = 5) => state
);

// change code below this line
```

Solution

```
const store = Redux.createStore(
  (state = 5) => state
);

// change code below this line
const currentState = store.getState();
```

3. Define a Redux Action

Description

Since Redux is a state management framework, updating state is one of its core tasks. In Redux, all state updates are triggered by dispatching actions. An action is simply a JavaScript object that contains information about an action event that has occurred. The Redux store receives these action objects, then updates its state accordingly. Sometimes a Redux action also carries some data. For example, the action carries a username after a user logs in. While the data is optional, actions must carry a `type` property that specifies the 'type' of action that occurred. Think of Redux actions as messengers that deliver information about events happening in your app to the Redux store. The store then conducts the business of updating state based on the action that occurred.

Instructions

Writing a Redux action is as simple as declaring an object with a type property. Declare an object `action` and give it a property `type` set to the string `'LOGIN'`.

Challenge Seed

```
// Define an action here:
```

Solution

```
const action = {
  type: 'LOGIN'
}
```

4. Define an Action Creator

Description

After creating an action, the next step is sending the action to the Redux store so it can update its state. In Redux, you define action creators to accomplish this. An action creator is simply a JavaScript function that returns an action. In other words, action creators create objects that represent action events.

Instructions

Define a function named `actionCreator()` that returns the `action` object when called.

Challenge Seed

```
const action = {
  type: 'LOGIN'
}
// Define an action creator here:
```

Solution

```
const action = {
  type: 'LOGIN'
}
// Define an action creator here:
const actionCreator = () => {
  return action;
};
```

5. Dispatch an Action Event

Description

`dispatch` method is what you use to dispatch actions to the Redux store. Calling `store.dispatch()` and passing the value returned from an action creator sends an action back to the store. Recall that action creators return an object with a type property that specifies the action that has occurred. Then the method dispatches an action object to the Redux store. Based on the previous challenge's example, the following lines are equivalent, and both dispatch the action of type `LOGIN`:

> store.dispatch(actionCreator());
> store.dispatch({ type: 'LOGIN' });

Instructions

The Redux store in the code editor has an initialized state that's an object containing a `login` property currently set to `false` . There's also an action creator called `loginAction()` which returns an action of type `LOGIN` . Dispatch the `LOGIN` action to the Redux store by calling the `dispatch` method, and pass in the action created by `loginAction()` .

Challenge Seed

```
const store = Redux.createStore(
  (state = {login: false}) => state
);

const loginAction = () => {
  return {
    type: 'LOGIN'
  }
};

// Dispatch the action here:
```

Solution

```
const store = Redux.createStore(
  (state = {login: false}) => state
);

const loginAction = () => {
  return {
    type: 'LOGIN'
  }
};

// Dispatch the action here:
store.dispatch(loginAction());
```

6. Handle an Action in the Store

Description

After an action is created and dispatched, the Redux store needs to know how to respond to that action. This is the job of a `reducer` function. Reducers in Redux are responsible for the state modifications that take place in response to actions. A `reducer` takes `state` and `action` as arguments, and it always returns a new `state` . It is important to see that this is the **only** role of the reducer. It has no side effects — it never calls an API endpoint and it never has any hidden surprises. The reducer is simply a pure function that takes state and action, then returns new state. Another key principle in Redux is that `state` is read-only. In other words, the `reducer` function must **always** return a new copy of `state` and never modify state directly. Redux does not enforce state immutability, however, you are responsible for enforcing it in the code of your reducer functions. You'll practice this in later challenges.

Instructions

The code editor has the previous example as well as the start of a `reducer` function for you. Fill in the body of the `reducer` function so that if it receives an action of type `'LOGIN'` it returns a state object with `login` set to `true` . Otherwise, it returns the current `state` . Note that the current `state` and the dispatched `action` are passed to the reducer, so you can access the action's type directly with `action.type` .

Challenge Seed

```
const defaultState = {
  login: false
};

const reducer = (state = defaultState, action) => {
  // change code below this line

  // change code above this line
};

const store = Redux.createStore(reducer);

const loginAction = () => {
  return {
    type: 'LOGIN'
  }
};
```

Solution

```
const defaultState = {
  login: false
};

const reducer = (state = defaultState, action) => {

  if (action.type === 'LOGIN') {
    return {login: true}
  }

  else {
    return state
  }

};

const store = Redux.createStore(reducer);

const loginAction = () => {
  return {
    type: 'LOGIN'
  }
};
```

7. Use a Switch Statement to Handle Multiple Actions

Description

You can tell the Redux store how to handle multiple action types. Say you are managing user authentication in your Redux store. You want to have a state representation for when users are logged in and when they are logged out. You represent this with a single state object with the property `authenticated` . You also need action creators that create actions corresponding to user login and user logout, along with the action objects themselves.

Instructions

The code editor has a store, actions, and action creators set up for you. Fill in the `reducer` function to handle multiple authentication actions. Use a JavaScript `switch` statement in the `reducer` to respond to different action events. This is a standard pattern in writing Redux reducers. The switch statement should switch over `action.type` and return the appropriate authentication state. **Note:** At this point, don't worry about state immutability, since it is small and simple in this example. For each action, you can return a new object — for example, `{authenticated: true}` . Also, don't forget to write a `default` case in your switch statement that returns the current `state` . This is important because once your app has multiple reducers, they are all run any time an action dispatch is made, even when the action isn't related to that reducer. In such a case, you want to make sure that you return the current `state` .

Challenge Seed

```
const defaultState = {
  authenticated: false
};

const authReducer = (state = defaultState, action) => {
  // change code below this line

  // change code above this line
};

const store = Redux.createStore(authReducer);

const loginUser = () => {
  return {
    type: 'LOGIN'
  }
};

const logoutUser = () => {
  return {
    type: 'LOGOUT'
  }
};
```

Solution

```
const defaultState = {
  authenticated: false
};

const authReducer = (state = defaultState, action) => {

  switch (action.type) {

    case 'LOGIN':
      return {
        authenticated: true
      }

    case 'LOGOUT':
      return {
        authenticated: false
      }

    default:
      return state;

  }

};

const store = Redux.createStore(authReducer);

const loginUser = () => {
  return {
    type: 'LOGIN'
  }
};

const logoutUser = () => {
  return {
    type: 'LOGOUT'
  }
};
```

8. Use const for Action Types

Description

A common practice when working with Redux is to assign action types as read-only constants, then reference these constants wherever they are used. You can refactor the code you're working with to write the action types as `const` declarations.

Instructions

Declare `LOGIN` and `LOGOUT` as `const` values and assign them to the strings `'LOGIN'` and `'LOGOUT'`, respectively. Then, edit the `authReducer()` and the action creators to reference these constants instead of string values. **Note:** It's generally a convention to write constants in all uppercase, and this is standard practice in Redux as well.

Challenge Seed

```
// change code below this line

// change code above this line

const defaultState = {
  authenticated: false
};

const authReducer = (state = defaultState, action) => {

  switch (action.type) {

    case 'LOGIN':
      return {
        authenticated: true
      }

    case 'LOGOUT':
      return {
        authenticated: false
      }

    default:
      return state;

  }

};

const store = Redux.createStore(authReducer);

const loginUser = () => {
  return {
    type: 'LOGIN'
  }
};

const logoutUser = () => {
  return {
    type: 'LOGOUT'
  }
};
```

Solution

```
const LOGIN = 'LOGIN';
const LOGOUT = 'LOGOUT';

const defaultState = {
  authenticated: false
};

const authReducer = (state = defaultState, action) => {

  switch (action.type) {
    case LOGIN:
      return {
        authenticated: true
      }

    case LOGOUT:
      return {
        authenticated: false
      }

    default:
      return state;

  }

};

const store = Redux.createStore(authReducer);

const loginUser = () => {
  return {
    type: LOGIN
  }
};

const logoutUser = () => {
  return {
    type: LOGOUT
  }
};
```

9. Register a Store Listener

Description

Another method you have access to on the Redux `store` object is `store.subscribe()` . This allows you to subscribe listener functions to the store, which are called whenever an action is dispatched against the store. One simple use for this method is to subscribe a function to your store that simply logs a message every time an action is received and the store is updated.

Instructions

Write a callback function that increments the global variable `count` every time the store receives an action, and pass this function in to the `store.subscribe()` method. You'll see that `store.dispatch()` is called three times in a row, each time directly passing in an action object. Watch the console output between the action dispatches to see the updates take place.

Challenge Seed

```
const ADD = 'ADD';

const reducer = (state = 0, action) => {
  switch(action.type) {
    case ADD:
      return state + 1;
    default:
      return state;
  }
};

const store = Redux.createStore(reducer);

// global count variable:
let count = 0;

// change code below this line
// change code above this line

store.dispatch({type: ADD});
console.log(count);
store.dispatch({type: ADD});
console.log(count);
store.dispatch({type: ADD});
console.log(count);
```

Before Test

```
count = 0;
```

Solution

```
const ADD = 'ADD';

const reducer = (state = 0, action) => {
  switch(action.type) {
    case ADD:
      return state + 1;
    default:
      return state;
  }
};

const store = Redux.createStore(reducer);
 let count = 0;
// change code below this line

store.subscribe( () =>
 {
 count++;
 }
);

// change code above this line

store.dispatch({type: ADD});
store.dispatch({type: ADD});
store.dispatch({type: ADD});
```

10. Combine Multiple Reducers

Description

When the state of your app begins to grow more complex, it may be tempting to divide state into multiple pieces. Instead, remember the first principle of Redux: all app state is held in a single state object in the store. Therefore, Redux provides reducer composition as a solution for a complex state model. You define multiple reducers to handle different pieces of your application's state, then compose these reducers together into one root reducer. The root reducer is then passed into the Redux `createStore()` method. In order to let us combine multiple reducers together, Redux provides the `combineReducers()` method. This method accepts an object as an argument in which you define properties which associate keys to specific reducer functions. The name you give to the keys will be used by Redux as the name for the associated piece of state. Typically, it is a good practice to create a reducer for each piece of application state when they are distinct or unique in some way. For example, in a note-taking app with user authentication, one reducer could handle authentication while another handles the text and notes that the user is submitting. For such an application, we might write the `combineReducers()` method like this:

> const rootReducer = Redux.combineReducers({
>   auth: authenticationReducer,
>   notes: notesReducer
> });

Now, the key `notes` will contain all of the state associated with our notes and handled by our `notesReducer` . This is how multiple reducers can be composed to manage more complex application state. In this example, the state held in the Redux store would then be a single object containing `auth` and `notes` properties.

Instructions

There are `counterReducer()` and `authReducer()` functions provided in the code editor, along with a Redux store. Finish writing the `rootReducer()` function using the `Redux.combineReducers()` method. Assign `counterReducer` to a key called `count` and `authReducer` to a key called `auth` .

Challenge Seed

```
const INCREMENT = 'INCREMENT';
const DECREMENT = 'DECREMENT';

const counterReducer = (state = 0, action) => {
  switch(action.type) {
    case INCREMENT:
      return state + 1;
    case DECREMENT:
      return state - 1;
    default:
      return state;
  }
};

const LOGIN = 'LOGIN';
const LOGOUT = 'LOGOUT';

const authReducer = (state = {authenticated: false}, action) => {
  switch(action.type) {
    case LOGIN:
      return {
        authenticated: true
      }
    case LOGOUT:
      return {
        authenticated: false
      }
    default:
      return state;
  }
};

const rootReducer = // define the root reducer here

const store = Redux.createStore(rootReducer);
```

Solution

```
const INCREMENT = 'INCREMENT';
const DECREMENT = 'DECREMENT';

const counterReducer = (state = 0, action) => {
  switch(action.type) {
    case INCREMENT:
      return state + 1;
    case DECREMENT:
      return state - 1;
    default:
      return state;
  }
};

const LOGIN = 'LOGIN';
const LOGOUT = 'LOGOUT';

const authReducer = (state = {authenticated: false}, action) => {
  switch(action.type) {
    case LOGIN:
      return {
        authenticated: true
      }
    case LOGOUT:
      return {
        authenticated: false
      }
    default:
      return state;
  }
};

const rootReducer = Redux.combineReducers({
  count: counterReducer,
  auth: authReducer
});

const store = Redux.createStore(rootReducer);
```

11. Send Action Data to the Store

Description

By now you've learned how to dispatch actions to the Redux store, but so far these actions have not contained any information other than a `type` . You can also send specific data along with your actions. In fact, this is very common because actions usually originate from some user interaction and tend to carry some data with them. The Redux store often needs to know about this data.

Instructions

There's a basic `notesReducer()` and an `addNoteText()` action creator defined in the code editor. Finish the body of the `addNoteText()` function so that it returns an `action` object. The object should include a `type` property with a value of `ADD_NOTE` , and also a `text` property set to the `note` data that's passed into the action creator. When you call the action creator, you'll pass in specific note information that you can access for the object. Next, finish writing the `switch` statement in the `notesReducer()` . You need to add a case that handles the `addNoteText()` actions. This case should be triggered whenever there is an action of type `ADD_NOTE` and it should return the `text` property on the incoming `action` as the new `state` . The action is dispatched at the bottom of the code. Once you're finished, run the code and watch the console. That's all it takes to send action-specific data to the store and use it when you update store `state` .

Challenge Seed

```
const ADD_NOTE = 'ADD_NOTE';

const notesReducer = (state = 'Initial State', action) => {
  switch(action.type) {
    // change code below this line

    // change code above this line
    default:
      return state;
  }
};

const addNoteText = (note) => {
  // change code below this line

  // change code above this line
};

const store = Redux.createStore(notesReducer);

console.log(store.getState());
store.dispatch(addNoteText('Hello!'));
console.log(store.getState());
```

Solution

```
const ADD_NOTE = 'ADD_NOTE';

const notesReducer = (state = 'Initial State', action) => {
  switch(action.type) {
    // change code below this line
    case ADD_NOTE:
      return action.text;
    // change code above this line
    default:
      return state;
  }
};

const addNoteText = (note) => {
  // change code below this line
  return {
    type: ADD_NOTE,
    text: note
  }
  // change code above this line
};

const store = Redux.createStore(notesReducer);

console.log(store.getState());
store.dispatch(addNoteText('Hello Redux!'));
console.log(store.getState());
```

12. Use Middleware to Handle Asynchronous Actions

Description

So far these challenges have avoided discussing asynchronous actions, but they are an unavoidable part of web development. At some point you'll need to call asynchronous endpoints in your Redux app, so how do you handle these types of requests? Redux provides middleware designed specifically for this purpose, called Redux Thunk middleware. Here's a brief description how to use this with Redux. To include Redux Thunk middleware, you pass it as an argument to `Redux.applyMiddleware()`. This statement is then provided as a second optional parameter to the `createStore()` function. Take a look at the code at the bottom of the editor to see this. Then, to create an asynchronous action, you return a function in the action creator that takes `dispatch` as an argument. Within this function, you can dispatch actions and perform asynchronous requests. In this example, an asynchronous request is simulated with a `setTimeout()` call. It's common to dispatch an action before initiating any asynchronous behavior so that your application state knows that some data is being requested (this state could display a loading icon, for instance). Then, once you receive the data, you dispatch another action which carries the data as a payload along with information that the action is completed. Remember that you're passing `dispatch` as a parameter to this special action creator. This is what you'll use to dispatch your actions, you simply pass the action directly to dispatch and the middleware takes care of the rest.

Instructions

Write both dispatches in the `handleAsync()` action creator. Dispatch `requestingData()` before the `setTimeout()` (the simulated API call). Then, after you receive the (pretend) data, dispatch the `receivedData()` action, passing in this data. Now you know how to handle asynchronous actions in Redux. Everything else continues to behave as before.

Challenge Seed

```
const REQUESTING_DATA = 'REQUESTING_DATA'
const RECEIVED_DATA = 'RECEIVED_DATA'
const requestingData = () => { return {type: REQUESTING_DATA} }
const receivedData = (data) => { return {type: RECEIVED_DATA, users: data.users} }

const handleAsync = () => {
  return function(dispatch) {
    // dispatch request action here

    setTimeout(function() {
      let data = {
        users: ['Jeff', 'William', 'Alice']
      }
      // dispatch received data action here

    }, 2500);
  }
};

const defaultState = {
  fetching: false,
  users: []
};

const asyncDataReducer = (state = defaultState, action) => {
  switch(action.type) {
    case REQUESTING_DATA:
      return {
        fetching: true,
        users: []
      }
    case RECEIVED_DATA:
      return {
        fetching: false,
        users: action.users
      }
    default:
      return state;
  }
};

const store = Redux.createStore(
  asyncDataReducer,
  Redux.applyMiddleware(ReduxThunk.default)
);
```

Solution

```
const REQUESTING_DATA = 'REQUESTING_DATA'
const RECEIVED_DATA = 'RECEIVED_DATA'

const requestingData = () => { return {type: REQUESTING_DATA} }
const receivedData = (data) => { return {type: RECEIVED_DATA, users: data.users} }

const handleAsync = () => {
  return function(dispatch) {
    dispatch(requestingData());
    setTimeout(function() {
      let data = {
        users: ['Jeff', 'William', 'Alice']
      }
      dispatch(receivedData(data));
    }, 2500);
  }
};

const defaultState = {
  fetching: false,
  users: []
};

const asyncDataReducer = (state = defaultState, action) => {
  switch(action.type) {
    case REQUESTING_DATA:
      return {
        fetching: true,
        users: []
      }
    case RECEIVED_DATA:
      return {
        fetching: false,
        users: action.users
      }
    default:
      return state;
  }
};

const store = Redux.createStore(
  asyncDataReducer,
  Redux.applyMiddleware(ReduxThunk.default)
);
```

13. Write a Counter with Redux

Description

Now you've learned all the core principles of Redux! You've seen how to create actions and action creators, create a Redux store, dispatch your actions against the store, and design state updates with pure reducers. You've even seen how to manage complex state with reducer composition and handle asynchronous actions. These examples are simplistic, but these concepts are the core principles of Redux. If you understand them well, you're ready to start building your own Redux app. The next challenges cover some of the details regarding `state` immutability, but first, here's a review of everything you've learned so far.

Instructions

In this lesson, you'll implement a simple counter with Redux from scratch. The basics are provided in the code editor, but you'll have to fill in the details! Use the names that are provided and define `incAction` and `decAction` action creators, the `counterReducer()`, `INCREMENT` and `DECREMENT` action types, and finally the Redux `store`. Once you're finished you should be able to dispatch `INCREMENT` or `DECREMENT` actions to increment or decrement the state held in the `store`. Good luck building your first Redux app!

Challenge Seed

```
const INCREMENT = null; // define a constant for increment action types
const DECREMENT = null; // define a constant for decrement action types

const counterReducer = null; // define the counter reducer which will increment or decrement the state
based on the action it receives

const incAction = null; // define an action creator for incrementing

const decAction = null; // define an action creator for decrementing

const store = null; // define the Redux store here, passing in your reducers
```

Solution

```
const INCREMENT = 'INCREMENT';
const DECREMENT = 'DECREMENT';

const counterReducer = (state = 0, action) => {
  switch(action.type) {
    case INCREMENT:
      return state + 1;
    case DECREMENT:
      return state - 1;
    default:
      return state;
  }
};

const incAction = () => {
  return {
    type: INCREMENT
  }
};

const decAction = () => {
  return {
    type: DECREMENT
  }
};

const store = Redux.createStore(counterReducer);
```

14. Never Mutate State

Description

These final challenges describe several methods of enforcing the key principle of state immutability in Redux. Immutable state means that you never modify state directly, instead, you return a new copy of state. If you took a snapshot of the state of a Redux app over time, you would see something like `state 1`, `state 2`, `state 3`, `state 4`, `...` and so on where each state may be similar to the last, but each is a distinct piece of data. This immutability, in fact, is what provides such features as time-travel debugging that you may have heard about. Redux does not actively enforce state immutability in its store or reducers, that responsibility falls on the programmer. Fortunately, JavaScript (especially ES6) provides several useful tools you can use to enforce the immutability of your state, whether it is a `string`, `number`, `array`, or `object`. Note that strings and numbers are primitive values and are immutable by nature. In other words, 3 is always 3. You cannot change the value of the number 3. An `array` or `object`, however, is mutable. In practice, your state will probably consist of an `array` or `object`, as these are useful data structures for representing many types of information.

Instructions

There is a `store` and `reducer` in the code editor for managing to-do items. Finish writing the `ADD_TO_DO` case in the reducer to append a new to-do to the state. There are a few ways to accomplish this with standard JavaScript or ES6. See if you can find a way to return a new array with the item from `action.todo` appended to the end.

Challenge Seed

```
const ADD_TO_DO = 'ADD_TO_DO';

// A list of strings representing tasks to do:
const todos = [
  'Go to the store',
  'Clean the house',
  'Cook dinner',
  'Learn to code',
];

const immutableReducer = (state = todos, action) => {
  switch(action.type) {
    case ADD_TO_DO:
      // don't mutate state here or the tests will fail
      return
    default:
      return state;
  }
};

// an example todo argument would be 'Learn React',
const addToDo = (todo) => {
  return {
    type: ADD_TO_DO,
    todo
  }
}

const store = Redux.createStore(immutableReducer);
```

Solution

```
const ADD_TO_DO = 'ADD_TO_DO';

// A list of strings representing tasks to do:
const todos = [
  'Go to the store',
  'Clean the house',
  'Cook dinner',
  'Learn to code',
];

const immutableReducer = (state = todos, action) => {
  switch(action.type) {
    case ADD_TO_DO:
      return state.concat(action.todo);
    default:
      return state;
  }
};

// an example todo argument would be 'Learn React',
const addToDo = (todo) => {
  return {
    type: ADD_TO_DO,
    todo
  }
}

const store = Redux.createStore(immutableReducer);
```

15. Use the Spread Operator on Arrays

Description

One solution from ES6 to help enforce state immutability in Redux is the spread operator: `...` . The spread operator has a variety of applications, one of which is well-suited to the previous challenge of producing a new array from an existing array. This is relatively new, but commonly used syntax. For example, if you have an array `myArray` and write: `let newArray = [...myArray];` `newArray` is now a clone of `myArray` . Both arrays still exist separately in memory. If you perform a mutation like `newArray.push(5)` , `myArray` doesn't change. The `...` effectively *spreads* out the values in `myArray` into a new array. To clone an array but add additional values in the new array, you could write `[...myArray, 'new value']` . This would return a new array composed of the values in `myArray` and the string `'new value'` as the last value. The spread syntax can be used multiple times in array composition like this, but it's important to note that it only makes a shallow copy of the array. That is to say, it only provides immutable array operations for one-dimensional arrays.

Instructions

Use the spread operator to return a new copy of state when a to-do is added.

Challenge Seed

```
const immutableReducer = (state = ['Do not mutate state!'], action) => {
  switch(action.type) {
    case 'ADD_TO_DO':
      // don't mutate state here or the tests will fail
      return
    default:
      return state;
  }
};

const addToDo = (todo) => {
  return {
    type: 'ADD_TO_DO',
    todo
  }
}

const store = Redux.createStore(immutableReducer);
```

Solution

```
const immutableReducer = (state = ['Do not mutate state!'], action) => {
  switch(action.type) {
    case 'ADD_TO_DO':
      return [
        ...state,
        action.todo
      ];
    default:
      return state;
  }
};

const addToDo = (todo) => {
  return {
    type: 'ADD_TO_DO',
    todo
  }
}

const store = Redux.createStore(immutableReducer);
```

16. Remove an Item from an Array

Description

Time to practice removing items from an array. The spread operator can be used here as well. Other useful JavaScript methods include `slice()` and `concat()`.

Instructions

The reducer and action creator were modified to remove an item from an array based on the index of the item. Finish writing the reducer so a new state array is returned with the item at the specific index removed.

Challenge Seed

```
const immutableReducer = (state = [0,1,2,3,4,5], action) => {
  switch(action.type) {
    case 'REMOVE_ITEM':
      // don't mutate state here or the tests will fail
      return
    default:
      return state;
  }
};

const removeItem = (index) => {
  return {
    type: 'REMOVE_ITEM',
    index
  }
}

const store = Redux.createStore(immutableReducer);
```

Solution

```
const immutableReducer = (state = [0,1,2,3,4,5], action) => {
  switch(action.type) {
    case 'REMOVE_ITEM':
      return [
        ...state.slice(0, action.index),
        ...state.slice(action.index + 1)
      ];
    default:
      return state;
  }
};

const removeItem = (index) => {
  return {
    type: 'REMOVE_ITEM',
    index
  }
}

const store = Redux.createStore(immutableReducer);
```

17. Copy an Object with Object.assign

Description

The last several challenges worked with arrays, but there are ways to help enforce state immutability when state is an `object` , too. A useful tool for handling objects is the `Object.assign()` utility. `Object.assign()` takes a target object and source objects and maps properties from the source objects to the target object. Any matching properties are overwritten by properties in the source objects. This behavior is commonly used to make shallow copies of objects by passing an empty object as the first argument followed by the object(s) you want to copy. Here's an example: `const newObject = Object.assign({}, obj1, obj2);` This creates `newObject` as a new `object` , which contains the properties that currently exist in `obj1` and `obj2` .

Instructions

The Redux state and actions were modified to handle an `object` for the `state` . Edit the code to return a new `state` object for actions with type `ONLINE` , which set the `status` property to the string `online` . Try to use `Object.assign()` to complete the challenge.

Challenge Seed

```
const defaultState = {
  user: 'CamperBot',
  status: 'offline',
  friends: '732,982',
  community: 'freeCodeCamp'
};

const immutableReducer = (state = defaultState, action) => {
  switch(action.type) {
    case 'ONLINE':
      // don't mutate state here or the tests will fail
      return
    default:
      return state;
  }
};

const wakeUp = () => {
  return {
    type: 'ONLINE'
  }
};

const store = Redux.createStore(immutableReducer);
```

Solution

```
const defaultState = {
  user: 'CamperBot',
  status: 'offline',
  friends: '732,982',
  community: 'freeCodeCamp'
};

const immutableReducer = (state = defaultState, action) => {
  switch(action.type) {
    case 'ONLINE':
      return Object.assign({}, state, {
        status: 'online'
      });
    default:
      return state;
  }
};

const wakeUp = () => {
  return {
    type: 'ONLINE'
  }
};

const store = Redux.createStore(immutableReducer);
```

React and Redux

1. Getting Started with React Redux

Description

This series of challenges introduces how to use Redux with React. First, here's a review of some of the key principles of each technology. React is a view library that you provide with data, then it renders the view in an efficient, predictable way. Redux is a state management framework that you can use to simplify the management of your application's state. Typically, in a React Redux app, you create a single Redux store that manages the state of your entire app. Your React components subscribe to only the pieces of data in the store that are relevant to their role. Then, you dispatch actions directly from React components, which then trigger store updates. Although React components can manage their own state locally, when you have a complex app, it's generally better to keep the app state in a single location with Redux. There are exceptions when individual components may have local state specific only to them. Finally, because Redux is not designed to work with React out of the box, you need to use the `react-redux` package. It provides a way for you to pass Redux `state` and `dispatch` to your React components as `props` . Over the next few challenges, first, you'll create a simple React component which allows you to input new text messages. These are added to an array that's displayed in the view. This should be a nice review of what you learned in the React lessons. Next, you'll create a Redux store and actions that manage the state of the messages array. Finally, you'll use `react-redux` to connect the Redux store with your component, thereby extracting the local state into the Redux store.

Instructions

Start with a `DisplayMessages` component. Add a constructor to this component and initialize it with a state that has two properties: `input` , that's set to an empty string, and `messages` , that's set to an empty array.

Challenge Seed

```
class DisplayMessages extends React.Component {
  // change code below this line

  // change code above this line
  render() {
    return <div />
  }
};
```

After Test

```
ReactDOM.render(<DisplayMessages />, document.getElementById('root'))
```

Solution

```
class DisplayMessages extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      messages: []
    }
  }
  render() {
    return <div/>
  }
};
```

2. Manage State Locally First

Description

Here you'll finish creating the `DisplayMessages` component.

Instructions

First, in the `render()` method, have the component render an `input` element, `button` element, and `ul` element. When the `input` element changes, it should trigger a `handleChange()` method. Also, the `input` element should render the value of `input` that's in the component's state. The `button` element should trigger a `submitMessage()` method when it's clicked. Second, write these two methods. The `handleChange()` method should update the `input` with what the user is typing. The `submitMessage()` method should concatenate the current message (stored in `input`) to the `messages` array in local state, and clear the value of the `input` . Finally, use the `ul` to map over the array of `messages` and render it to the screen as a list of `li` elements.

Challenge Seed

```
class DisplayMessages extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      messages: []
    }
  }
```

```
    // add handleChange() and submitMessage() methods here

    render() {
      return (
        <div>
          <h2>Type in a new Message:</h2>
          { /* render an input, button, and ul here */ }

          { /* change code above this line */ }
        </div>
      );
    }
  };
```

After Test

```
ReactDOM.render(<DisplayMessages />, document.getElementById('root'))
```

Solution

```
class DisplayMessages extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      messages: []
    }
 this.handleChange = this.handleChange.bind(this);
   this.submitMessage = this.submitMessage.bind(this);
 }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  submitMessage() {
    const currentMessage = this.state.input;
    this.setState({
      input: '',
      messages: this.state.messages.concat(currentMessage)
    });
  }
  render() {
    return (
      <div>
        <h2>Type in a new Message:</h2>
        <input
          value={this.state.input}
          onChange={this.handleChange}/><br/>
        <button onClick={this.submitMessage}>Submit</button>
        <ul>
          {this.state.messages.map( (message, idx) => {
              return (
                 <li key={idx}>{message}</li>
              )
            })
          }
        </ul>
      </div>
    );
  }
};
```

3. Extract State Logic to Redux

Description

Now that you finished the React component, you need to move the logic it's performing locally in its `state` into Redux. This is the first step to connect the simple React app to Redux. The only functionality your app has is to add new messages from the user to an unordered list. The example is simple in order to demonstrate how React and Redux work together.

Instructions

First, define an action type 'ADD' and set it to a const `ADD` . Next, define an action creator `addMessage()` which creates the action to add a message. You'll need to pass a `message` to this action creator and include the message in the returned `action` . Then create a reducer called `messageReducer()` that handles the state for the messages. The initial state should equal an empty array. This reducer should add a message to the array of messages held in state, or return the current state. Finally, create your Redux store and pass it the reducer.

Challenge Seed

```
// define ADD, addMessage(), messageReducer(), and store here:
```

Solution

```
const ADD = 'ADD';

const addMessage = (message) => {
  return {
    type: ADD,
    message
  }
};

const messageReducer = (state = [], action) => {
  switch (action.type) {
    case ADD:
      return [
        ...state,
        action.message
      ];
    default:
      return state;
  }
};

const store = Redux.createStore(messageReducer);
```

4. Use Provider to Connect Redux to React

Description

In the last challenge, you created a Redux store to handle the messages array and created an action for adding new messages. The next step is to provide React access to the Redux store and the actions it needs to dispatch updates. React Redux provides its `react-redux` package to help accomplish these tasks. React Redux provides a small API with two key features: `Provider` and `connect` . Another challenge covers `connect` . The `Provider` is a wrapper component from React Redux that wraps your React app. This wrapper then allows you to access the Redux `store` and `dispatch` functions throughout your component tree. `Provider` takes two props, the Redux store and the child components of your app. Defining the `Provider` for an App component might look like this:

> <Provider store={store}>
>   <App/>
> </Provider>

Instructions

The code editor now shows all your Redux and React code from the past several challenges. It includes the Redux store, actions, and the `DisplayMessages` component. The only new piece is the `AppWrapper` component at the bottom. Use this top level component to render the `Provider` from `ReactRedux` , and pass the Redux store as a prop. Then render the `DisplayMessages` component as a child. Once you are finished, you should see your React component rendered to the page. **Note:** React Redux is available as a global variable here, so you can access the Provider with dot notation. The code in the editor takes advantage of this and sets it to a constant `Provider` for you to use in the `AppWrapper` render method.

Challenge Seed

```
// Redux Code:
const ADD = 'ADD';

const addMessage = (message) => {
  return {
    type: ADD,
    message
  }
};

const messageReducer = (state = [], action) => {
  switch (action.type) {
    case ADD:
      return [
        ...state,
        action.message
      ];
    default:
      return state;
  }
};



const store = Redux.createStore(messageReducer);

// React Code:

class DisplayMessages extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      messages: []
    }
    this.handleChange = this.handleChange.bind(this);
    this.submitMessage = this.submitMessage.bind(this);
  }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  submitMessage() {
    const currentMessage = this.state.input;
    this.setState({
      input: '',
      messages: this.state.messages.concat(currentMessage)
    });
  }
  render() {
    return (
      <div>
        <h2>Type in a new Message:</h2>
        <input
          value={this.state.input}
          onChange={this.handleChange}/><br/>
        <button onClick={this.submitMessage}>Submit</button>
        <ul>
          {this.state.messages.map( (message, idx) => {
              return (
                 <li key={idx}>{message}</li>
              )
            })
          }
        </ul>
      </div>
    );
  }
};

const Provider = ReactRedux.Provider;

class AppWrapper extends React.Component {
  // render the Provider here

  // change code above this line
};
```

After Test

```
ReactDOM.render(<AppWrapper />, document.getElementById('root'))
```

Solution

```
// Redux Code:
const ADD = 'ADD';

const addMessage = (message) => {
  return {
    type: ADD,
    message
  }
};

const messageReducer = (state = [], action) => {
  switch (action.type) {
    case ADD:
      return [
        ...state,
        action.message
      ];
    default:
      return state;
  }
};

const store = Redux.createStore(messageReducer);

// React Code:

class DisplayMessages extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      messages: []
    }
 this.handleChange = this.handleChange.bind(this);
 this.submitMessage = this.submitMessage.bind(this);
 }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  submitMessage() {
    const currentMessage = this.state.input;
    this.setState({
      input: '',
      messages: this.state.messages.concat(currentMessage)
    });
  }
  render() {
    return (
      <div>
        <h2>Type in a new Message:</h2>
        <input
          value={this.state.input}
          onChange={this.handleChange}/><br/>
        <button onClick={this.submitMessage}>Submit</button>
        <ul>
          {this.state.messages.map( (message, idx) => {
              return (
                 <li key={idx}>{message}</li>
              )
            })
          }
        </ul>
      </div>
    );
  }
};

const Provider = ReactRedux.Provider;

class AppWrapper extends React.Component {
  // change code below this line
  render() {
    return (
      <Provider store = {store}>
        <DisplayMessages/>
      </Provider>
    );
  }
  // change code above this line
};
```

5. Map State to Props

Description

The `Provider` component allows you to provide `state` and `dispatch` to your React components, but you must specify exactly what state and actions you want. This way, you make sure that each component only has access to the state it needs. You accomplish this by creating two functions: `mapStateToProps()` and `mapDispatchToProps()` . In these functions, you declare what pieces of state you want to have access to and which action creators you need to be able to dispatch. Once these functions are in place, you'll see how to use the React Redux `connect` method to connect them to your components in another challenge. **Note:** Behind the scenes, React Redux uses the `store.subscribe()` method to implement `mapStateToProps()` .

Instructions

Create a function `mapStateToProps()` . This function should take `state` as an argument, then return an object which maps that state to specific property names. These properties will become accessible to your component via `props` . Since this example keeps the entire state of the app in a single array, you can pass that entire state to your component. Create a property `messages` in the object that's being returned, and set it to `state` .

Challenge Seed

```
const state = [];

// change code below this line
```

Solution

```
const state = [];

// change code below this line
```

```
const mapStateToProps = (state) => {
  return {
    messages: state
  }
};
```

6. Map Dispatch to Props

Description

The `mapDispatchToProps()` function is used to provide specific action creators to your React components so they can dispatch actions against the Redux store. It's similar in structure to the `mapStateToProps()` function you wrote in the last challenge. It returns an object that maps dispatch actions to property names, which become component `props`. However, instead of returning a piece of `state`, each property returns a function that calls `dispatch` with an action creator and any relevant action data. You have access to this `dispatch` because it's passed in to `mapDispatchToProps()` as a parameter when you define the function, just like you passed `state` to `mapStateToProps()`. Behind the scenes, React Redux is using Redux's `store.dispatch()` to conduct these dispatches with `mapDispatchToProps()`. This is similar to how it uses `store.subscribe()` for components that are mapped to `state`. For example, you have a `loginUser()` action creator that takes a `username` as an action payload. The object returned from `mapDispatchToProps()` for this action creator would look something like:

> {
> submitLoginUser: function(username) {
>   dispatch(loginUser(username));
> }
> }

Instructions

The code editor provides an action creator called `addMessage()`. Write the function `mapDispatchToProps()` that takes `dispatch` as an argument, then returns an object. The object should have a property `submitNewMessage` set to the dispatch function, which takes a parameter for the new message to add when it dispatches `addMessage()`.

Challenge Seed

```
const addMessage = (message) => {
  return {
    type: 'ADD',
    message: message
  }
};

// change code below this line
```

Solution

```
const addMessage = (message) => {
  return {
    type: 'ADD',
    message: message
  }
};

// change code below this line

const mapDispatchToProps = (dispatch) => {
  return {
    submitNewMessage: function(message) {
      dispatch(addMessage(message));
    }
  }
};
```

7. Connect Redux to React

Description

Now that you've written both the `mapStateToProps()` and the `mapDispatchToProps()` functions, you can use them to map `state` and `dispatch` to the `props` of one of your React components. The `connect` method from React Redux can handle this task. This method takes two optional arguments, `mapStateToProps()` and `mapDispatchToProps()`. They are optional because you may have a component that only needs access to `state` but doesn't need to dispatch any actions, or vice versa. To use this method, pass in the functions as arguments, and immediately call the result with your component. This syntax is a little unusual and looks like: `connect(mapStateToProps, mapDispatchToProps)(MyComponent)` **Note:** If you want to omit one of the arguments to the `connect` method, you pass `null` in its place.

Instructions

The code editor has the `mapStateToProps()` and `mapDispatchToProps()` functions and a new React component called `Presentational`. Connect this component to Redux with the `connect` method from the `ReactRedux` global object, and call it immediately on the `Presentational` component. Assign the result to a new `const` called `ConnectedComponent` that represents the connected component. That's it, now you're connected to Redux! Try changing either of `connect`'s arguments to `null` and observe the test results.

Challenge Seed

```
const addMessage = (message) => {
  return {
    type: 'ADD',
    message: message
  }
};

const mapStateToProps = (state) => {
  return {
    messages: state
  }
};

const mapDispatchToProps = (dispatch) => {
  return {
    submitNewMessage: (message) => {
      dispatch(addMessage(message));
    }
  }
};

class Presentational extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <h3>This is a Presentational Component</h3>
  }
};

const connect = ReactRedux.connect;
// change code below this line
```

After Test

```
const store = Redux.createStore(
  (state = '__INITIAL__STATE__', action) => state
);
```

```
class AppWrapper extends React.Component {
  render() {
    return (
      <ReactRedux.Provider store = {store}>
        <ConnectedComponent/>
      </ReactRedux.Provider>
    );
  }
};
ReactDOM.render(<AppWrapper />, document.getElementById('root'))
```

Solution

```
const addMessage = (message) => {
  return {
    type: 'ADD',
    message: message
  }
};

const mapStateToProps = (state) => {
  return {
    messages: state
  }
};

const mapDispatchToProps = (dispatch) => {
  return {
    submitNewMessage: (message) => {
      dispatch(addMessage(message));
    }
  }
};

class Presentational extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <h3>This is a Presentational Component</h3>
  }
};

const connect = ReactRedux.connect;
// change code below this line

const ConnectedComponent = connect(mapStateToProps, mapDispatchToProps)(Presentational);
```

8. Connect Redux to the Messages App

Description

Now that you understand how to use `connect` to connect React to Redux, you can apply what you've learned to your React component that handles messages. In the last lesson, the component you connected to Redux was named `Presentational`, and this wasn't arbitrary. This term *generally* refers to React components that are not directly connected to Redux. They are simply responsible for the presentation of UI and do this as a function of the props they receive. By contrast, container components are connected to Redux. These are typically responsible for dispatching actions to the store and often pass store state to child components as props.

Instructions

The code editor has all the code you've written in this section so far. The only change is that the React component is renamed to `Presentational`. Create a new component held in a constant called `Container` that uses `connect` to connect the `Presentational` component to Redux. Then, in the `AppWrapper`, render the React Redux `Provider` component. Pass `Provider` the Redux `store` as a prop and render `Container` as a child. Once everything is setup, you will see the messages app rendered to the page again.

Challenge Seed

```
// Redux:
const ADD = 'ADD';

const addMessage = (message) => {
  return {
    type: ADD,
    message: message
  }
};

const messageReducer = (state = [], action) => {
  switch (action.type) {
    case ADD:
      return [
        ...state,
        action.message
      ];
    default:
      return state;
  }
};

const store = Redux.createStore(messageReducer);

// React:
class Presentational extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      messages: []
    }
    this.handleChange = this.handleChange.bind(this);
    this.submitMessage = this.submitMessage.bind(this);
  }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  submitMessage() {
    const currentMessage = this.state.input;
    this.setState({
      input: '',
      messages: this.state.messages.concat(currentMessage)
    });
  }
  render() {
    return (
      <div>
        <h2>Type in a new Message:</h2>
        <input
          value={this.state.input}
          onChange={this.handleChange}/><br/>
        <button onClick={this.submitMessage}>Submit</button>
        <ul>
          {this.state.messages.map( (message, idx) => {
              return (
                 <li key={idx}>{message}</li>
              )
            })
          }
        </ul>
      </div>
    );
  }
};

// React-Redux:
const mapStateToProps = (state) => {
  return { messages: state }
};

const mapDispatchToProps = (dispatch) => {
  return {
    submitNewMessage: (newMessage) => {
       dispatch(addMessage(newMessage))
    }
  }
};

const Provider = ReactRedux.Provider;
const connect = ReactRedux.connect;

// define the Container component here:


class AppWrapper extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    // complete the return statement:
    return (null);
  }
};
```

After Test

```
ReactDOM.render(<AppWrapper />, document.getElementById('root'))
```

Solution

```
// Redux:
const ADD = 'ADD';

const addMessage = (message) => {
  return {
    type: ADD,
    message: message
  }
};

const messageReducer = (state = [], action) => {
  switch (action.type) {
    case ADD:
      return [
        ...state,
        action.message
      ];
    default:
      return state;
  }
};

const store = Redux.createStore(messageReducer);

// React:
class Presentational extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      messages: []
    }
 this.handleChange = this.handleChange.bind(this);
 this.submitMessage = this.submitMessage.bind(this);
  }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  submitMessage() {
    const currentMessage = this.state.input;
    this.setState({
      input: '',
      messages: this.state.messages.concat(currentMessage)
    });
  }
  render() {
    return (
      <div>
        <h2>Type in a new Message:</h2>
        <input
          value={this.state.input}
          onChange={this.handleChange}/><br/>
        <button onClick={this.submitMessage}>Submit</button>
        <ul>
          {this.state.messages.map( (message, idx) => {
              return (
                 <li key={idx}>{message}</li>
              )
            })
          }
        </ul>
      </div>
    );
  }
};

// React-Redux:
const mapStateToProps = (state) => {
  return { messages: state }
};

const mapDispatchToProps = (dispatch) => {
  return {
    submitNewMessage: (newMessage) => {
       dispatch(addMessage(newMessage))
    }
  }
};

const Provider = ReactRedux.Provider;
const connect = ReactRedux.connect;

// define the Container component here:
const Container = connect(mapStateToProps, mapDispatchToProps)(Presentational);

class AppWrapper extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    // complete the return statement:
    return (
      <Provider store={store}>
        <Container/>
      </Provider>
    );
  }
};
```

9. Extract Local State into Redux

Description

You're almost done! Recall that you wrote all the Redux code so that Redux could control the state management of your React messages app. Now that Redux is connected, you need to extract the state management out of the `Presentational` component and into Redux. Currently, you have Redux connected, but you are handling the state locally within the `Presentational` component.

Instructions

In the `Presentational` component, first, remove the `messages` property in the local `state`. These messages will be managed by Redux. Next, modify the `submitMessage()` method so that it dispatches `submitNewMessage()` from `this.props`, and pass in the current message input from local `state` as an argument. Because you removed `messages` from local state, remove the `messages` property from the call to `this.setState()` here as well. Finally, modify the `render()` method so that it maps over the messages received from `props` rather than `state`. Once these changes are made, the app will continue to function the same, except Redux manages the state. This example also illustrates how a component may have local `state`: your component still tracks user input locally in its own `state`. You can see how Redux provides a useful state management framework on top of React. You achieved the same result using only React's local state at first, and this is usually possible with simple apps. However, as your apps become larger and more complex, so does your state management, and this is the problem Redux solves.

Challenge Seed

```
// Redux:
const ADD = 'ADD';

const addMessage = (message) => {
  return {
    type: ADD,
    message: message
  }
};

const messageReducer = (state = [], action) => {
  switch (action.type) {
    case ADD:
      return [
        ...state,
        action.message
      ];
    default:
      return state;
  }
};

const store = Redux.createStore(messageReducer);

// React:
const Provider = ReactRedux.Provider;
const connect = ReactRedux.connect;

// Change code below this line
class Presentational extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      messages: []
    }
    this.handleChange = this.handleChange.bind(this);
    this.submitMessage = this.submitMessage.bind(this);
  }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  submitMessage() {
    this.setState({
      input: '',
      messages: this.state.messages.concat(this.state.input)
    });
  }
  render() {
    return (
      <div>
        <h2>Type in a new Message:</h2>
        <input
          value={this.state.input}
          onChange={this.handleChange}/><br/>
        <button onClick={this.submitMessage}>Submit</button>
        <ul>
          {this.state.messages.map( (message, idx) => {
              return (
                 <li key={idx}>{message}</li>
              )
            })
          }
        </ul>
      </div>
    );
  }
};
// Change code above this line

const mapStateToProps = (state) => {
  return {messages: state}
};

const mapDispatchToProps = (dispatch) => {
  return {
    submitNewMessage: (message) => {
      dispatch(addMessage(message))
    }
  }
};

const Container = connect(mapStateToProps, mapDispatchToProps)(Presentational);

class AppWrapper extends React.Component {
  render() {
    return (
      <Provider store={store}>
        <Container/>
      </Provider>
    );
  }
};
```

After Test

```
ReactDOM.render(<AppWrapper />, document.getElementById('root'))
```

Solution

```
// Redux:
const ADD = 'ADD';

const addMessage = (message) => {
  return {
    type: ADD,
    message: message
  }
};

const messageReducer = (state = [], action) => {
  switch (action.type) {
    case ADD:
      return [
        ...state,
        action.message
      ];
    default:
      return state;
  }
};

const store = Redux.createStore(messageReducer);

// React:
const Provider = ReactRedux.Provider;
const connect = ReactRedux.connect;
```

```
// Change code below this line
class Presentational extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: ''
    }
 this.handleChange = this.handleChange.bind(this);
 this.submitMessage = this.submitMessage.bind(this);
  }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  submitMessage() {
    this.props.submitNewMessage(this.state.input);
    this.setState({
      input: ''
    });
  }
  render() {
    return (
      <div>
        <h2>Type in a new Message:</h2>
        <input
          value={this.state.input}
          onChange={this.handleChange}/><br/>
        <button onClick={this.submitMessage}>Submit</button>
        <ul>
          {this.props.messages.map( (message, idx) => {
              return (
                 <li key={idx}>{message}</li>
              )
            })
          }
        </ul>
      </div>
    );
  }
};
// Change code above this line

const mapStateToProps = (state) => {
  return {messages: state}
};

const mapDispatchToProps = (dispatch) => {
  return {
    submitNewMessage: (message) => {
      dispatch(addMessage(message))
    }
  }
};

const Container = connect(mapStateToProps, mapDispatchToProps)(Presentational);

class AppWrapper extends React.Component {
  render() {
    return (
      <Provider store={store}>
        <Container/>
      </Provider>
    );
  }
};
```

10. Moving Forward From Here

Description

Congratulations! You finished the lessons on React and Redux. There's one last item worth pointing out before you move on. Typically, you won't write React apps in a code editor like this. This challenge gives you a glimpse of what the syntax looks like if you're working with npm and a file system on your own machine. The code should look similar, except for the use of `import` statements (these pull in all of the dependencies that have been provided for you in the challenges). The "Managing Packages with npm" section covers npm in more detail. Finally, writing React and Redux code generally requires some configuration. This can get complicated quickly. If you are interested in experimenting on your own machine, the Create React App comes configured and ready to go. Alternatively, you can enable Babel as a JavaScript Preprocessor in CodePen, add React and ReactDOM as external JavaScript resources, and work there as well.

Instructions

Log the message `'Now I know React and Redux!'` to the console.

Challenge Seed

```
// import React from 'react'
// import ReactDOM from 'react-dom'
// import { Provider, connect } from 'react-redux'
// import { createStore, combineReducers, applyMiddleware } from 'redux'
// import thunk from 'redux-thunk'

// import rootReducer from './redux/reducers'
// import App from './components/App'

// const store = createStore(
//   rootReducer,
//   applyMiddleware(thunk)
// );

// ReactDOM.render(
//   <Provider store={store}>
//     <App/>
//   </Provider>,
//   document.getElementById('root')
// );

// change code below this line
```

Solution

```
console.log('Now I know React and Redux!');
```

Front End Libraries Projects

1. Build a Random Quote Machine

Description

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/qRZeGZ. Fulfill the below user stories and get all of the tests to pass. Give it your own personal style. You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding! **User Story #1:** I can see a wrapper element with a corresponding `id="quote-box"` . **User Story #2:** Within `#quote-box` , I can see an element with a corresponding `id="text"` . **User Story #3:** Within `#quote-box` , I can see an element with a corresponding `id="author"` . **User Story #4:** Within `#quote-box` , I can see a clickable element with a corresponding `id="new-quote"` .

User Story #5: Within `#quote-box` , I can see a clickable element with a corresponding `id="tweet-quote"` . **User Story #6:** On first load, my quote machine displays a random quote in the element with `id="text"` . **User Story #7:** On first load, my quote machine displays the random quote's author in the element with `id="author"` . **User Story #8:** When the `#new-quote` button is clicked, my quote machine should fetch a new quote and display it in the `#text` element. **User Story #9:** My quote machine should fetch the new quote's author when the `#new-quote` button is clicked and display it in the `#author` element. **User Story #10:** I can tweet the current quote by clicking on the `#tweet-quote` `a` element. This `a` element should include the `"twitter.com/intent/tweet"` path in its `href` attribute to tweet the current quote. **User Story #11:** The `#quote-box` wrapper element should be horizontally centered. Please run tests with browser's zoom level at 100% and page maximized. You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: `https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js` Once you're done, submit the URL to your working project with all its tests passing. Remember to use the Read-Search-Ask method if you get stuck.

Instructions

Challenge Seed

Solution

```
// solution required
```

2. Build a Markdown Previewer

Description

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/GrZVVO. Fulfill the below user stories and get all of the tests to pass. Give it your own personal style. You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding! **User Story #1:** I can see a `textarea` element with a corresponding `id="editor"` . **User Story #2:** I can see an element with a corresponding `id="preview"` . **User Story #3:** When I enter text into the `#editor` element, the `#preview` element is updated as I type to display the content of the textarea. **User Story #4:** When I enter GitHub flavored markdown into the `#editor` element, the text is rendered as HTML in the `#preview` element as I type (HINT: You don't need to parse Markdown yourself - you can import the Marked library for this: https://cdnjs.com/libraries/marked). **User Story #5:** When my markdown previewer first loads, the default text in the `#editor` field should contain valid markdown that represents at least one of each of the following elements: a header (H1 size), a sub header (H2 size), a link, inline code, a code block, a list item, a blockquote, an image, and bolded text. **User Story #6:** When my markdown previewer first loads, the default markdown in the `#editor` field should be rendered as HTML in the `#preview` element. **Optional Bonus (you do not need to make this test pass):** My markdown previewer interprets carriage returns and renders them as `br` (line break) elements. You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: `https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js` Once you're done, submit the URL to your working project with all its tests passing. Remember to use the Read-Search-Ask method if you get stuck.

Instructions

Challenge Seed

Solution

```
// solution required
```

3. Build a Drum Machine

Description

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/MJyNMd. Fulfill the below user stories and get all of the tests to pass. Give it your own personal style. You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding! **User Story #1:** I should be able to see an outer container with a corresponding `id="drum-machine"` that contains all other elements. **User Story #2:** Within `#drum-machine` I can see an element with a corresponding `id="display"` . **User Story #3:** Within `#drum-machine` I can see 9 clickable drum pad elements, each with a class name of `drum-pad` , a unique id that describes the audio clip the drum pad will be set up to trigger, and an inner text that corresponds to one of the following keys on the keyboard: Q, W, E, A, S, D, Z, X, C. The drum pads MUST be in this order. **User Story #4:** Within each `.drum-pad` , there should be an HTML5 `audio` element which has a `src` attribute pointing to an audio clip, a class name of `clip` , and an id corresponding to the inner text of its parent `.drum-pad` (e.g. `id="Q"` , `id="W"` , `id="E"` etc.). **User Story #5:** When I click on a `.drum-pad` element, the audio clip contained in its child `audio` element should be triggered. **User Story #6:** When I press the trigger key associated with each `.drum-pad` , the audio clip contained in its child `audio` element should be triggered (e.g. pressing the Q key should trigger the drum pad which contains the string "Q", pressing the W key should trigger the drum pad which contains the string "W", etc.). **User Story #7:** When a `.drum-pad` is triggered, a string describing the associated audio clip is displayed as the inner text of the `#display` element (each string must be unique). You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: `https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js` Once you're done, submit the URL to your working project with all its tests passing. Remember to use the Read-Search-Ask method if you get stuck.

Instructions

Challenge Seed

Solution

```
// solution required
```

4. Build a JavaScript Calculator

Description

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/wgGVVX. Fulfill the below user stories and get all of the tests to pass. Give it your own personal style. You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding! **User Story #1:** My calculator should contain a clickable element containing an `=` (equal sign) with a corresponding `id="equals"` . **User Story #2:** My calculator should contain 10 clickable elements containing one number each from 0-9, with the following corresponding IDs: `id="zero"` , `id="one"` , `id="two"` , `id="three"` , `id="four"` , `id="five"` , `id="six"` , `id="seven"` , `id="eight"` , and `id="nine"` . **User Story #3:** My calculator should contain 4 clickable elements each containing one of the 4 primary mathematical operators with the following corresponding IDs: `id="add"` , `id="subtract"` , `id="multiply"`, `id="divide"`. **User Story #4:** My calculator should contain a clickable element containing a `.` (decimal point) symbol with a corresponding `id="decimal"`. **User Story #5:** My calculator should contain a clickable element with an `id="clear"`. **User Story #6:** My calculator should contain an element to display values with a corresponding `id="display"`. **User Story #7:** At any time, pressing the clear button clears the input and output values, and returns the calculator to its initialized state; 0 should be shown in the element with the id of `display`. **User Story #8:** As I input numbers, I should be able to see my input in the element with the id of `display`. **User Story #9:** In any order, I should be able to add, subtract, multiply and divide a chain of numbers of any length, and when I hit `=`, the correct result should be shown in the element with the id of `display`. **User Story #10:** When inputting numbers, my calculator should not allow a number to begin with multiple zeros. **User Story #11:** When the decimal element is clicked, a `.` should append to the currently displayed value; two `.` in one number should not be accepted. **User Story #12:** I should be able to perform any operation (+, -, *, /) on numbers containing decimal points. **User Story #13:** If 2 or more operators are entered consecutively, the operation performed should be the last operator entered. **User Story #14:** Pressing an operator immediately following `=` should start a new calculation that operates on the result of the previous evaluation. **User Story #15:** My calculator should have several decimal places of precision when it comes to rounding (note that there is no exact standard, but you should be able to handle calculations like `2 / 7` with reasonable precision to at least 4 decimal places). **Note On Calculator Logic:** It should be noted that there are two main schools of thought on calculator input logic: immediate execution logic and formula logic. Our example utilizes formula logic and observes order of operation precedence, immediate execution does not. Either is acceptable, but please note that depending on which you choose, your calculator may yield different results than ours for certain equations (see below example). As long as your math can be verified by another production calculator, please do not consider this a bug. **EXAMPLE:** `3 + 5 x 6 - 2 / 4 =`

- **Immediate Execution Logic:** `11.5`
- **Formula/Expression Logic:** `32.5`

You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: `https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js` Once you're done, submit the URL to your working project with all its tests passing. Remember to use the Read-Search-Ask method if you get stuck.

Instructions

Challenge Seed

Solution

```
// solution required
```

5. Build a Pomodoro Clock

Description

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/XpKrrW. Fulfill the below user stories and get all of the tests to pass. Give it your own personal style. You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding! **User Story #1:** I can see an element with `id="break-label"` that contains a string (e.g. "Break Length"). **User Story #2:** I can see an element with `id="session-label"` that contains a string (e.g. "Session Length"). **User Story #3:** I can see two clickable elements with corresponding IDs: `id="break-decrement"` and `id="session-decrement"`. **User Story #4:** I can see two clickable elements with corresponding IDs: `id="break-increment"` and `id="session-increment"`. **User Story #5:** I can see an element with a corresponding `id="break-length"`, which by default (on load) displays a value of 5. **User Story #6:** I can see an element with a corresponding `id="session-length"`, which by default displays a value of 25. **User Story #7:** I can see an element with a corresponding `id="timer-label"`, that contains a string indicating a session is initialized (e.g. "Session"). **User Story #8:** I can see an element with corresponding `id="time-left"`. NOTE: Paused or running, the value in this field should always be displayed in `mm:ss` format (i.e. 25:00). **User Story #9:** I can see a clickable element with a corresponding `id="start_stop"`. **User Story #10:** I can see a clickable element with a corresponding `id="reset"`. **User Story #11:** When I click the element with the id of `reset`, any running timer should be stopped, the value within `id="break-length"` should return to `5`, the value within `id="session-length"` should return to 25, and the element with `id="time-left"` should reset to it's default state. **User Story #12:** When I click the element with the id of `break-decrement`, the value within `id="break-length"` decrements by a value of 1, and I can see the updated value. **User Story #13:** When I click the element with the id of `break-increment`, the value within `id="break-length"` increments by a value of 1, and I can see the updated value. **User Story #14:** When I click the element with the id of `session-decrement`, the value within `id="session-length"` decrements by a value of 1, and I can see the updated value. **User Story #15:** When I click the element with the id of `session-increment`, the value within `id="session-length"` increments by a value of 1, and I can see the updated value. **User Story #16:** I should not be able to set a session or break length to <= 0. **User Story #17:** I should not be able to set a session or break length to > 60. **User Story #18:** When I first click the element with `id="start_stop"`, the timer should begin running from the value currently displayed in `id="session-length"`, even if the value has been incremented or decremented from the original value of 25. **User Story #19:** If the timer is running, the element with the id of `time-left` should display the remaining time in `mm:ss` format (decrementing by a value of 1 and updating the display every 1000ms). **User Story #20:** If the timer is running and I click the element with `id="start_stop"`, the countdown should pause. **User Story #21:** If the timer is paused and I click the element with `id="start_stop"`, the countdown should resume running from the point at which it was paused. **User Story #22:** When a session countdown reaches zero (NOTE: timer MUST reach 00:00), and a new countdown begins, the element with the id of `timer-label` should display a string indicating a break has begun. **User Story #23:** When a session countdown reaches zero (NOTE: timer MUST reach 00:00), a new break countdown should begin, counting down from the value currently displayed in the `id="break-length"` element. **User Story #24:** When a break countdown reaches zero (NOTE: timer MUST reach 00:00), and a new countdown begins, the element with the id of `timer-label` should display a string indicating a session has begun. **User Story #25:** When a break countdown reaches zero (NOTE: timer MUST reach 00:00), a new session countdown should begin, counting down from the value currently displayed in the `id="session-length"` element. **User Story #26:** When a countdown reaches zero (NOTE: timer MUST reach 00:00), a sound indicating that time is up should play. This should utilize an HTML5 `audio` tag and have a corresponding `id="beep"`. **User Story #27:** The audio element with `id="beep"` must be 1 second or longer. **User Story #28:** The audio element with id of `beep` must stop playing and be rewound to the beginning when the element with the id of `reset` is clicked. You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: `https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js` Once you're done, submit the URL to your working project with all its tests passing. Remember to use the Read-Search-Ask method if you get stuck.

Instructions

Challenge Seed

Solution

```
// solution required
```