Quiz #2 Solutions

Question 1

Question 1.1

1. React requires state changes to be made using this.setState() so it can easily find components whose state has changed. Proper coding would look like:
   ```
   this.setState({count: this.state.count + 1}
   ```
2. Updating the this.state without using setState() means React will not notice the state change and will not call re-render the component. The user-visible behavior will be a click on the buttons do not cause the number in the button to change.

Question 1.2

The correct answer was F, and the correct reason was rather than passing the function this.handleClick to onClick, the code calls the function and passed the return value to onClick. Since the return value of the this.handleClick isn't a function this won't do anything good. Calling this.handleClick during the execution of render() will also fail since React doesn't like to see setState() calls during render(). You could only get full credit by selecting F, although some students selected F *and* also correctly explained why other answers were redundant or bad style, which was fine. Some partial credit was awarded to students who:

- Correctly selected F, but with an incomplete justification.
- Correctly selected and justified F, but incorrectly selected 1 or 2 other choices.
- Failed to select F, but correctly explained what's bad about a different choice.

Students who made multiple incorrect selections, or made incorrect selections *and* failed to identify the problem with F, received no credit.

Question 1.3

The error is caused by an attempt to interpret inline JSX (specifically, a `<div>`) as JavaScript, which fails because JSX is not JavaScript. The fundamental error is that the browser cannot handle JSX, and it has to be transpiled into pure JavaScript before it can be used by the browser. There were many variations of this that received full credit.

Partial credit was given to answers that did not get at the fundamental problem, but perhaps suggested a technical fix (such as adding `type="text/babel"` to the script tag and using `@babel/standalone`), or otherwise got close (e.g. by saying "the browser expects JavaScript but we gave it HTML", which is close, but doesn't get to the heart of the mistaken approach).

Partial credit was also given to answers that identified the basic problem correctly, but made false assertions as part of their explanation, such as "the browser is trying to interpret this as HTML", or "the error happens because `<` is an illegal token". (The symbol < is certainly not an illegal token in inline JavaScript, as you can verify for yourself by checking `if (2 < 3)`.)

Other answers received no credit.

Question 2

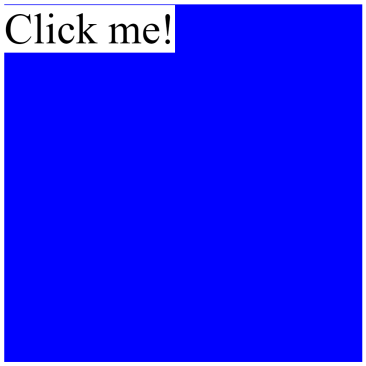
The "top or surface" of deep linking is the initial view of the web application. Without deep linking, you can not enter the web app anywhere but the initial view of the web app. With deep linking, you can get beyond the initial view and show the view described in the link.

Full credit was given for answers that identified initial view under alternative names like "home page or hostname URL.

Partial credit was given for ill-defined terms for the "top or surface" like "the URL" or "app context". No credit was given for answers simply defining what deep linking is.

Question 3

Before we start contemplating event listeners, note that the four lines setting the style of the objects in the DOM would lead us to expect to see a white backgrounded text saying “Click me!” surrounded by blue background:



Question 3.1

Clicking on the blue region around the button entails clicking on `container`. We have registered two event listeners on this region. Both are the function `one`, and the first is for the capture phase and the second for the bubble phase. Thus, we should expect the function `one` to be called twice. `event.target` is the element that triggered the event, which is `container`. `event.currentTarget` is the element that the event listener is registered on, which is also `container`. Thus, they are the same, and we should expect the console to print

ONE
ONE

Note that because `button` is not clicked, its even listeners will not be triggered.

Question 3.2

Clicking on the “Click me!” entails clicking on `button` and also clicking on `container`, given that `container` subsumes `button`. Thus, in the capture phase, where we go from outermost element to innermost, we trigger the event listener registered on `container`. Because in this case, `event.target`, the element that triggered the event, is button, but `event.currentTarget` is `container`, this will print

one

Then, in the bubble phase, we first trigger the event listener registered on `button`, which will print

TWO

because both `event.target` and `event.currentTarget` are `button`. Following this, the capture phase event listener on `container` will trigger, printing

one

Question 4

```
<div id=“myDiv”>
    <p class="cs142-style-0">CS142</p>
    <div class="cs142-style-1">Quiz2</div>
    <a class="cs142-style-2" href="http://localhost:3000">Link</a>
</div>
```

Question 5

Prior to browser-based JavaScript web app frameworks, the server would need to render the view into HTML to the display by the browser. In the MVC pattern, that meant combining the view and model data parts together was done the server now is being done in the browser. Instead of having to all the work of rendering views (e.g filling in templates, etc.) and the run-time support for doing that, web servers simply serve static parts of the web app and serve model data fetches.

Question 6

Since this was a "Select all that apply." problem we need to examine and accept or reject each of the claims:

Without the real-time capabilities of WebSockets, we can’t expect the changes to be pushed to people’s browsers within a day of being updated.

> Even using plain HTTP requests, a fetch of the web app contents would return the latest copy of the web app. No special WebSockets-based protocol is needed for that to happen. This wouldn't explain the difficulties.

Mendel may be using a “stateful” rather than a “stateless” web server, which usually causes the server to send clients the same version of the site they loaded previously.

> The terms "stateful" and "stateless", when used for web servers, is talking about session state. Getting old versions of a web app doesn't really have much to do with this.

The student may have a cached version of the website, which her computer loaded rather than fetching the newest version.

> Caching of content in the browser (i.e. Cache-Control - HTTP header) could explain this behavior. Access to the web app content being returned from the browser's cache rather than the web server would show up as the old version of the web app show.

Question 7

To rewrite to not use `await` you need to convert the await into a `.then` call on the promise. The remaining part of the async function is put in the callback to `.then`

Question 7.1

The remaining part of the function after the await simply adds one so a clean way to write the answer is:

```
function afunc1() {
    return p1().then((x) => {x + 1});
}
```

p1() returns a promise and we access the value by calling then(). In the then(), we want to increment the value from p1() and return it as a promise (in the scope of the then()). Finally, we want to return the returned promise from the then() as the return value of afunc1().

Solutions that created a new promise with 'new Promise((resolve, reject)=>{})' needed to properly handle the rejected cases so that the user of afunc1() can handle rejected promises.

Question 7.2

There were multiple ways to solve this. Below is one way:

```
function afunc2() {
    return p1().then((p1Val) => {
        if (p1Val & 1) {
            return p2().then((p2Val) => {
                return p2Val + 1;
            });
        } else {
            return p2().then((p2Val) => {
                return p3().then((p3Val) => p3Val + p2Val);    });
        }
    });
}
```

Because the final return value of afunc2() is a promise, you needed to correctly return promises in the nested then() statements. Keep in mind that in arrow functions, if the only thing inside of the function body is a single expression, it automatically returns the result of that expression.

Again, solutions that created a new promise with `new Promise((resolve, reject)=>{})` needed to properly handle the rejected cases so that the user of afunc2() can handle rejected promises.

Question 7.3

`afunc1()` and `afunc2()` are asynchronous and return promises. When doing an assignment like `let z = afunc1()`, `afunc1()` will immediately return with an unresolved promise and assign it to z. This would take only a few microseconds so the closest number of whole seconds would be 0 for both `afunc1()` and `afunc2()`.

Question 8

RESTful APIs use GET methods to fetch data from the server. POST, PUT, or DELETE methods are used to update resources. Since this form is adding a resource (a tweet) the form should a POST method to be REST-like.

Question 9

The console.log statement will output 0. fs.readFile is an asynchronous function, so the console.log statement after the fs.readFile call will be executed before fs.readFile calls its done callback. At the point the console.log is executed `fileContents` will be the value it is initialized to (an empty object) so Object.keys will return an empty array of keys.

Question 10

Node.js uses a single-threaded event loop. Everything runs as a call from this event loop. If the current event has a blocking operation(such as file and socket/network io), it is wrapped in the event interface and asynchronously dispatched to the OS (Node.js has access to a thread pool in the OS). The function for the current event, can then immediately return, and the event loop will pick out the next event from the queue and start processing that one as the asynchronous operation continues in the background. When the asynchronous operation is complete, it creates a new event with the callback function specified at creation, and inserts that new event into the back of the event queue/loop.

Requests being processed are broken into these event handlers and handlers from many active requests can be in the event queue at a given time. The processing of a request's handlers can be interleaved with handlers of other requests. This gives a sense of concurrency even though only one handler is ever running at a time. For credit, we were looking for a clear understanding of the use of an event loop, asynchronous operations/functions which allow concurrency (blocking vs non-blocking), and callback functions.

Question 11

The correct output (along with a note explaining the reason) is shown below:

Output from `.emit("foo")`:

`Hello, world!` - foo emitted first, only one listener

Output from `.emit("bar", 1, 3)`:

4 - First bar listener is called, so result is += (3 + 1)

16 - Second bar listener is then called, so result *= (2 + 3 - 1)

Output from `.emit("who")`:

- Nothing is outputted. If no listener then emit() is a nop.

Note: "error" listener is only called when eventEmitter emits “`error`”