# Quiz #2 Solutions

## Question 1

### *1.1*

```
let element = document.body.firstChild;
try { console.log(element.id); } catch (err) { console.log('ERROR')}
```

The body starts with:

```
<body>
   Body
 <div id="div1"
…
```

So the `firstChild` of body is the text node ("Body").   It doesn't have an id property so the best answer would be None of the above.

### *1.2*

```
let element = document.getElementById('div3').parentNode;
try { console.log(element.id); } catch (err) { console.log('ERROR')}
```

'div3' is located inside of `<div id="div2" ….`
So the parentNode of div3 would be div2 so the best answer would be 'div2'

### *1.3*

```
let element = document.getElementsByClassName('c2');
try { console.log(element[0].id); } catch (err) { console.log('ERROR')}
```

The getElementsByClassName on the document will return a.l the elements with class="c2".  The first and only of which is the div with id='div5'.  The answer should be div5.

### *1.4*

```
let element = document.getElementById('div3').getElementsByClassName('c2');
try { console.log(element[0].id); } catch (err) { console.log('ERROR')}
```

Would do a looking for class="c2" starting at 'div3' in the DOM tree.  Since div5 isn't in the subtree under 'div2' it would return an empty array.  Element 0 would be undefined so we would get an exception through.  Best answer would be ERROR.

### *1.5*

```
let element = document.getElementById('div3');
```

```
try {element.style.display = "none" } catch (err) { console.log('ERROR')}
```

The assignment would set display:none on div3 so div3 and it children will disappear. That would leave Body, Div1, Div2, Div5 displaying in the window.

### *1.6*

```
document.getElementById('div3').style.position = "absolute";
```

Would switch 'div3' to be position absolutely in the current positioning context.  It turns how that the current positing context is the body of the document so the absolution is the same as the current top and left of the element.

## **Question 2**

We have both capture and bubble phase click handers so the click would fire the handlers in the capture phase starting at the root and going down.  Next it would run the bubble phase going up.  Each hander prints:

| | |
|---|---|
| e.currentTarget.nodeName | - The HTML tag of DOM node with the handler. |
| e.currentTarget.id, | - The id attribute of DOM node with the handle (nothing if there is no id) |
| E.target.id | - The id clicked on (div3) |

```
Capture html  div3
Capture body  div3
Capture div div2 div3
Capture div div3 div3
Bubble div div3 div3
Bubble div div2 div3
Bubble body  div3
Bubble html  div3
```

## **Question 3**

The four lines of code do:

1. Prints log1 to log
2. Register the function printing log2 to fire 10ms
3. Prints log3 to log
4. Register the function printing log4 to fire in 1ms

The expected output order would be:

```
log1, log3, log4, log2
```

## Question 4

### *4.1*

The equivalence of a promise resolving to a number would be just a callback being passed the number. So

```
function One(doneCallback) {
    doneCallback(1);
}
function Two(doneCallback) {
    doneCallback(2);
}
```

### *4.2*

This one is much more difficult. We know it would need to add the doneCallback to the Sum function so it would look like:

```
function SumC(op1, op2, doneCallback) { …. /* Call doneCallback with val*/ }
```

The promise version computes the return value (`await op1 + await op1`) by resolving op1 and summing it with the resolved op2. Since op1 and op2 return values with callbacks we would pass a function into it. We can resolve op1 and then op2 to get the values to sum and call the doneCallback with.

```
function SumC(op1, op2, doneCallback) {
   op1((op1Val) => {
          op2((op2Val) => doneCallback(op1Val + op2Val));
        });
}
```

## Question 5

### *5.1*

The JavaScript runtime environment runs all code as functions that must return before some other function runs. This run to completion approach means any individual JavaScript statement's execution can not be interleaved with any other execution making the race conditions of multithreaded environments like Java and C++ not an issue. There is only a single thread of control running JavaScript code.

### *5.2*

Calling a long running function like the listed Sync operations means the currently running function will not complete until the operation over. Every other function on the event queue will need to wait until the current one completes. This effectively stops all execution of JavaScript code.

The effect on the web app depends on what the app uses JavaScript execution to do. The React web app we use push view into the DOM so the browser does the rendering but most user interaction uses event handlers that now would be delayed waiting for the sync operation to complete.  The end result would be a web app that is slow to respond or do anything of the things that require JavaScript execution that receiving data from the network or user, etc.

## **Question 6**

JSX is a preprocessor that outputs JavaScript containing calls to React.createElement.

```
<div id="div1"></div>
```

Would output a call to React.createElement with the first argument being a 'div' and the attributes having id="div1".   More exactly:

```
React.createElement("div", {id: "div1"});
```

## **Question 7**

### *7.1*

When a React component is constructed there is nothing else looking at the component state so the JavaScript code in the constructor can directly manipulate `this.state` with normal JavaScript assignment statements. After a component is constructed and mounted React needs to track any changes to the `props` or `this.state` so it knows when it needs to call the component `render` method to get the updated virtual DOM tree returned by `render`.   This is part of a key performance optimization in React where components that don't change just continue to use the old value returned by render. `this.setState` provides this notification to React that a component state has changed.

If you were to assign directly to `this.state` and not use `this.setState()` the component state would be updated but React wouldn't know about it and would continue using the old value of the render() function.  The component view that the user sees wouldn't be updated to reflect these new state values unless something else triggered the reredering like a change in the props or something in the component lifecycle (e.g. unmount/remount).

### *7.2*

If we do:

```
this.state.counter = this.state.counter  + 1;
console.log(this.state.counter)
```

we are simply setting a javascript object property ("counter") and logging it to the console log. We'd expect the output to be `this.state.count + 1`.  If we do:

```
this.setState({counter: this.state.counter + 1}};
console.log(this.state.counter);
```

we are telling React we want the counter state updated. This is a delayed operation in React so the console log on the next lines wouldn't see the update.  It would log `this.state.counter`

## **Question 8**

In GraphQL, the user can specify exactly which properties they want from the large number of properties. With a REST system, even if the user only wanted 1 property, they would have to fetch the entire resource, which would result in a lot of unneeded data being transferred.

## **Question 9**

Correctly explaining the use of end-to-end testing will get you full credit. Using both is also acceptable as long as you justify why.

Since end-to-end testing is running the entire web application any responsive design features like CSS breakpoints could be examined with end-to-end tests. You just need to be able to run tests with different window/screen sizes.

Although unit testing focuses on individual components that might not have responsive design features in them to test, it would be possible to test a component that had CSS breakpoints assuming you had sufficiently detailed mocks in your unit testing environment things like the @media functionality is exposed during the unit test. In that way, you could have unit tests of CSS breakpoint behavior.

## **Question 10**

It is desirable to have a single page application (SPA) behave like a multiple-page website with respect to operations that users do while surfing the web such as bookmarking pages and sharing URLs of pages. The concept of deep linking allows a SPA to do this.

You should mention at least 2 of these points:

- Users are used to multi-page experiences and expect it
- Deep-linking, bookmarking, etc
- It enables back button and browser history functionalities
- It can be used to provide state/context to the application

## **Question 11**

HTTP is a request-response protocol meaning that all communication is initialized by the browser sending a request and the backend sending a response to that request. That makes communication between the browser and web server simple. The browser sends an HTTP request when it wants to communicate.  The web server, on the other, can only respond to requests from the browser. Any communication it wants to do must be part of some response to a request.