CS 142 Midterm Examination

Winter Quarter 2023

You have 1.5 hours (90 minutes) for this examination; the number of points for each question indicates roughly how many minutes you should spend on that question. Make sure you print your name and sign the Honor Code below. During the examination you may consult two double-sided pages of notes; all other sources of information, including laptops, cell phones, etc. are prohibited.

I acknowledge and accept the Stanford University Honor Code. I have neither given nor received aid in answering the questions on this examination.

_____

(Signature)

Solutions

_____

(Print your name, legibly!)

_____@stanford.edu

(Stanford email account for grading database key)

Problem #1 (6 points)

The word "hyper" in Hypertext Markup Language (HTML) gets its name from the ability to embed entities where the user's mouse clicks generate actions. For example, the HTML "a" tag renders text so a different document is viewed when the user clicks on the text.
(e.g. `<a href="clickhappened">click me</a>`)

With event handling, arbitrary JavaScript code including displaying a different document can be bound to a mouse action so any HTML tag can be "hyper". For example, a user clicks on a "b" tag or "p" tag can have a mouse click switch to a different view:
(e.g. `<p onclick="do_clickhappened()">click me</p>`)

Although with JavaScript frameworks like React.js, we could generate HTML that never used the "a" tag but instead used event handling to switch the HTML document being rendered by the browser. Explain why a developer using React.js might prefer to use the old HTML “a” tag and keep the browser involved in the view switching rather than doing the view switching solely in JavaScript.

Having the browser involved in view transitions formed the motivation for the deep linking idea in single page applications. The user is used to having the browser handle the view switching with hyperlinks, bookmarks, etc. so it is a natural extension to continue the view switching that way inside the web application. Down that the level of HTML, with "a" tags, the user will see the normal browser view transition mechanism (e.g. links highlight when moused over) that wouldn't be present if is another element with a click handler.

Problem #2 (8 points)

You are given the following HTML document that contains a placeholder (**XXX**) that can be set to any string:

```
<html>
   <head>
      <style type="text/css">
      p { position: XXX; top: 0; left: 0; }
      </style>
   </head>
   <body>
      <p>Paragraph 1</p>
      <p>Paragraph 2</p>
      <p>Paragraph 3</p>
    </body>
</html>
```

Describe how the HTML document would look when rendered with XXX set to the following values:

A. `"static"`
B. `"absolute"`
C. `"relative"`
D. `"fixed"`

A. `"static"` - The default on the browser the paragraphs will be displayed in document flow order. It would look like this:

Paragraph 1

Paragraph 2

Paragraph 3

B. `"absolute"` Each of the paragraphs will be positioned at (0,0) relative to the positioning context, which is the body. The paragraphs will be on top of each other in the top left of the body. It would look like this:

Paragraph 1

C. `"relative"` - Each of the paragraphs at (0,0) offset (no difference) from the offset of the document flow. It would be the same as the "static" case.

D. `"fixed"` - Each of the paragraphs will be positioned (0,0) relative to the viewport, which would likely be the same as the body. This means it would look the same as the "absolute" case.

Problem #3 (8 points)

Assume we have a React.js web application that was served from the URL `http://host/a.html` and contains the following generated hyperlinks:

```
<a href="#/user/list">Show users</a>
<a href="#full">Show full list</a>
```

Also, assume JavaScript click handling in the React.js runtime has broken so only the normal browser handling of "a" tags occurs. Assume the user clicked twice on the initial view. The first click was on "Show users" and they noticed no view changes (recall the JavaScript is broken). The second click was on "Show full list".

A. Describe what the browser processing would have been triggered by the first click (think about the processing the browser does on the click).
B. Show what the resulting location URL would be after the first click.
C. Show what the resulting location URL would be after the second click.

A. With JavaScript broken, the browser would handle the click by adding the fragment `#/user/list` to the URL. This would cause the browser to scan the DOM looking for an anchor tag with the matching `#/user/list`. Since the anchor tag isn't typically found in ReactJS applications, the view would be left unchanged.

B. With the added fragment, the URL location would be set to `http://localhost:3000/a.html#/user/list`

C. The second click would change the replace the fragment part of the URL with `#full` so the URL will be: `http://localhost:3000/a.html#full`

Problem #4 (7 points)

Consider the following ECMAScript code:

```
class Foo {
  methodA() { }
  static staticA() { }
}
let f = new Foo();
```

List all the JavaScript objects created by the code. For each object you list:

- Provide a JavaScript expression that accesses the object.
- State what properties (if any) the object contains.

`Foo` - The class keyword would create a function name Foo which is also an object. The object would contain the static method (`staticA`).

`Foo.prototype` - The Foo function would have a prototype object for instances. The methods (`methodA`) will be in that object.

`f` - The new keyword allocates an object instance of the Foo class. This object doesn't have any own properties.

Functions are objects in JavaScript and the above code create two additional functions:

`Foo.prototype.methodA` - This object doesn't have any own properties.

`Foo.staticA` - This object doesn't have any own properties.

Problem #5 (6 points)

In real-world math: `1 + 2 = 3`,

so by dividing both sides by `n`, we get that

```
1/n + 2/n = 3/n
```

will be true for all `n` != 0.

In JavaScript, it is true for many, but not all, values of `n`. Looking at the first 10 non-zero integers, the expression `1/n + 2/n == 3/n` evaluates to `false` for 20% of the numbers.

A. Explain why JavaScript arithmetic only gets 80% correct here.
B. If we changed the expression to `1/n + 2/n === 3/n` would there be any improvement in the result? Explain your answer.

A. JavaScript stores numbers using IEEE 64-bit floating point. Floating point numbers are stored as sign, fraction, and magnitude values. When doing arithmetic on non-integral values care needs to be taken that the arithmetic might not be exact and can be off by a small amount. In the first 10 non-zero integers, the n values of 5 and 10 return false for the expression due to these small errors.

B. The strict equality (===) operator checks for equality without doing any type conversion. Since both sides of the equations are the same type (Number) the === will return the same value as ==. Hence it will get the same number "incorrect".

Problem #6 (6 points)

Early JavaScript code conventionally used a variable name `self` when coding object-oriented programming methods containing callback functions. Explain how the introduction of shorthand for defining functions pushed by the advocates of functional programming (i.e. arrow function expressions) caused object-oriented programing developers to use the variable `self` less?

The pattern in JavaScript was to assign the variable `self` to point at the instance object like so:

```
let self = this;
```

so to have a regular variable (`self`) and can access this even inside of embedded function definitions. With the addition of arrow function expressions and don't redefine `this`, the code can use `this` inside embedded functions and there is no need to create an alias for `this`.

Problem #7 (6 points)

Assume you have the following HTML document:

```
<html id="id0">
   <body id="id1">
       <div id="id2">
          Div 1
          <div id="id3">
              Div 2
         </div>
       </div>
   </body>
</html>
```

Assume you looked up the DOM nodes by ID (i.e. `getElementById`) and recorded the IDs of the `offsetParent` and `parentNode`. Fill in the following table:

	`offsetParent.id`	`parentNode.id`
`id1`	body `offsetParent` is NULL by spec	`id0`
`id2`	`id1`	`id1`
`id3`	`id1`	`id2`

There is no additional positioning context set so the offsetParent of all the nodes inside of the body will be the id of the body (id1). The parentNode always points to the parent so id1 -> id0, and so on.

Problem #8 (8 points)

Assume you have the following HTML document (it's the same as Problem #7):

```
<html id="id0">
   <body id="id1">
       <div id="id2">
          Div 1
          <div id="id3">
              Div 2
         </div>
       </div>
   </body>
</html>
```

Assume you registered the same event handler on DOM nodes with IDs of `id0, id1, id2, id3` for both the capture and bubble phases (i.e., a total of 8 calls to `addEventListener`) You click on the line "Div 2" in the document. The event handler prints out the following:

1. Event phase ("bubble" or "capture")
2. `event.target.id`
3. `event.currentTarget.id`

In the table below, show what handler call prints you would expect to see. (Leave row blank if there are fewer calls than rows)

	Phase	`event.target.id`	`event.currentTarget.id`
1	"capture"	id3	id0
2	"capture"	id3	id1
3	"capture"	id3	id2
4	"capture"	id3	id3
5	"bubble"	id3	id3
6	"bubble"	id3	id2
7	"bubble"	id3	id1
8	"bubble"	id3	id0

The capture phase starts at the top (body) and goes down to the clicked element (id3). The bubble phase goes back up the DOM tree. The target will always be the clicked element (id3) and the currentTarget follows the phases down and up the tree.

Problem #9 (6 points)

HTML templates are commonly used for view generation in modern web application frameworks. These frameworks allow arbitrary complex expressions including function definitions and many line expressions to be embedded inside templates but good programming practices argue against long expressions. Explain what advantages of HTML templates are hurt by using long expressions.

Two of the advantages of HTML templates listed in the lecture notes are:
- Easy to visualize HTML structure
- Easy to see how dynamic data fits in

Have long dynamic data expressions undermind these advantages. If the expressions fill your editor window it becomes hard to see the HTML structure.

Problem #10 (8 points)

React.js permits you to directly read component state (e.g. `this.state.foobar`) but strongly discourages you from directly writing to component state (e.g. `this.state.foobar = newvalue`) once a component has been created.

A. Explain the rationale behind this asymmetry.
B. What misbehavior would you expect to happen if you ignored this suggestion? Use the model, view, controller (MVC) pattern to describe the problem.

A. React needs to efficiently detect changes to component state so it can optimize rendering performance by only re-rendering components whose state or props change. By having the user tell React every time a component's state changes, this detection can be done with low overhead. Forcing the developer to update state with a React wrapper function (`setState`) React can easily ignore components that don't need to be re-rendered and do only necessary work.

B. If you change a component state without using `setState` React will not "see" the change so the component will not be re-rendered with the updated state. The user will see the out-of-date view displayed.

Problem #11 (9 points)

In functional programming, it is typically required that a function's return value only depend on the input parameters being passed to the function and that the function doesn't change anything in the global state (i.e. no side-effects). When building a React.js component, we can create some state by using a call to `useState` from React Hooks like so:

```
let [val, setVal] = useState("init value");
```

A. React.js's `useState` can sometimes violate the functional programming requirement described above. Describe what would cause `useState` to return a different value for the same input parameter.
B. `useState` returns a JavaScript array. JavaScript supports arrays that are sparse and polymorphic. State if `useState` is using each of the following capabilities of JavaScript arrays. Briefly explain your answer showing your knowledge of the capability.
    a. sparse arrays
    b. polymorphic arrays

A. When `useState` is called the first time, it returns a state variable with value equal to the initial value passed in to `useState`. However, if the updating function (i.e., `setVal` in the above example) is called with a different value, the state variable returned from `useState` will now contain that new value. This explains how `useState` has side-effects.

B.
    a. Sparse arrays have fewer elements than their length. `useState` returns a JavaScript array that has two elements and is of length 2, so it is not using a sparse array.
    b. Polymorphic arrays hold values of different types. The element values returned by `useState` are the current value and the state updating function. Unless the state happens to be a function type, the two elements will be of different types. In the above example, the current value is a string which is a different type than the updating function. Thus, it does use polymorphic arrays.

Problem #12 (6 points)

Many modern web applications will dynamically update themselves if you change the size of the browser window they are running in. If you start the web application in a large window and slowly make the window smaller and smaller you notice the web application shrinks the amount of white space until spacing gets tight and then changes so less content is being displayed. It repeats this pattern of white space compression and content removal as the window gets smaller.

Describe the underlying browser mechanisms that make this white space compression and content removal behavior possible.

The browser's CSS `flexbox` and `grid` capabilities make it easy for web applications to specify components that are positioned to fill the available space. A shrinking window will be filled in with less space between the components. The CSS `@media` queries can be used to determine the window size and select what content should be displayed using CSS breakpoints. This mechanism can result in content being removed as the window gets smaller.

Problem #13 (6 points)

Would you expect to get better test coverage when using all end-2-end tests or all unit tests? Explain your answer.

Test coverage is a metric that measures the fraction of the total lines of code executed when tests are run. Better test coverage would be a high fraction with the ideal being tests execute 100% of the lines of code.

With unit tests, each "unit" in the system has tests written against it. Mocks are used to handle dependencies on other independent test units. It should be possible to get to 100% coverage with unit tests since any line of code that can't be executed by unit tests that can try all possible inputs can safely be deleted from the system.

End-2-end tests run the complete system and tests are done by simulating user input to a browser. Although end-2-end tests can detect problems that occur between units it is not necessarily possible to generate arbitrary inputs for every "unit' in the system. Less than 100% test coverage can occur.

Unit tests would expect better to achieve better test coverage.