# CSE 331
# Software Design & Implementation

Spring 2023

Section 8 – Midterm Review

# Administrivia

- Midterm on Friday (5/19) during usual class time
- HW8 released on late Friday/Saturday (5/19)
  - Due Friday (5/26) @ 11:00pm

# Stateful UI in React – Review

- UI so far has been static (Made UI changes by reloading page)
  - index.tsx was calls render to show a fixed UI

```
type HiProps = {name: string};
type HiState = {currName: string};
class HiElem extends Component<HiProps, HiState> {
     constructor(props: HiProps) {
          super(props);
          this.state = {currName: this.props.name};
     }
     setName = (newName: string): void => {
          this.setState({currName: newName});
     }
     render = (): JSX.Element {
          return <p> Hi, {this.state.currName}</p>;
     }
}
```

(readonly)

Initial value set in the constructor.
NEVER directly modified after

- Must call *setState* to change the state (React will automatically re-render when state changes)
- *Render* can use both `this.props` and `this.state`

# Event Handler – Review

- **Pass method to be called as argument**
  - **value of `onClick` attribute is our `makeSpanish` method**

```
render = (): JSX.Element {
   return (<div>
       <p>{this.state.greeting}, {this.props.name}!</p>
       <button onClick={this.makeSpanish}>Espanol</button>
     </div>);
 };
```

- **Browser will invoke that method when button is clicked**

```
makeSpanish = (evt: MouseEvent<HTMLButtonElement>) => {
   this.setState({greeting: "Hola"});
 };
```

  - **Call to `setState` causes a re-render (in a bit)**

# React Reminders – Review

- Make sure you declare your methods this way

```
onClick = (evt: MouseEvent<HTMLButtonElement) => {…};
```

- Note that *setState* is not instant
  - It adds an event that later updates the state (React tries to batch multiple updates together)
- Any state on the screen must be stored in some state
  - Text in any INPUT element must be in some state (ex: buttons, textboxes, etc.)
- NEVER modify anything in render
- NEVER modify *this.state* outside of constructor
  - Use *this.setState()* instead

# Question 1-3

```
git clone https://gitlab.cs.washington.edu/cse331-23sp-materials/sec-squares.git
```

Make sure to run ***npm install --no-audit*** in both the ***server*** directory and ***client*** directory. Then run ***npm start*** in both directories

# Definitions for Homework

**type** BST := empty
| node($x : \mathbb{Z}$, $S$ : BST, $T$ : BST) with conditions A and B

Suppose that we wanted to have a way to refer to a specific node in a BST. One way to do so would be to give directions from the root to that node. If we define these types:

**type** Dir := S | T
**type** Path := $\text{List}_{\text{Dir}}$

then a Path tells you how to get to a particular node. For example, cons($S$, cons($T$, nil)) says to select the "$S$" child of the parent and then the "$T$" child of that node, giving us a grand-child of the root node.

# Midterm Review

# Writing Loops – Midterm Review

Given the following loop invariant, fill in the body of the code that reverses an array in place. For your convenience, you can use a function `swap(arr: number[], i: number, j: number)` that takes in an array `arr` and swaps the elements at `i` and `j`.

Hint: Remember that swap changes the elements at BOTH indexes. How does that affect the exit condition?

```
function reverseArray(s: number[]): void {
  const n: number = s.length;
  let i: number = ____;

  {{ s[0…i-1] = rev(s_0[n-i…n-1]) and s[n-i .. n-1] = rev(s0[0 .. i-1]) }}
  while (__________) {

        i = i + 1;
  }
  {{ s[0…s.length-1] = rev(s_0[0…s.length-1]) }}
}
```

# Reasoning – Review

Below is an implementation of a non so efficient sorting algorithm, insertion sort. Fill in the missing assertions where $P_i$'s are from forward reasoning and $Q_i$'s are from backward reasoning. Then, prove that $P_i$ implies $Q_i$ for i = 1, 2, 3, 4

```
function insertionSort(A: number[]): void {
    let i: number = 0;
    {{ P1:_____ }}
    {{ Inv1: A[k] <= A[k+1] for any 0 <= k < i-1 }}
    while ( i !== A.length ) {
        let j: number = i - 1;
        let val: number = A[i];
        {{ P2: ____________________________________________ }}
        {{ Inv2: val = A[j + 1] and A[j + 1] <=… <= A[i] and A[0] <= A[1] <= … <= A[j - 1] <= A[j] }}
        while ( j !== -1 && A[j] > val ) {
            {{ P3: ___________________________ }}
            {{ Q3: ____________________________________________________________________________}}
            A[j + 1] = A[j];
            A[j] = val;
            j = j - 1;
        }
        {{ P4: ____________________________________________________ }}
        {{ Q4: _____________________ }}
        i = i + 1;
    }
    {{ P5: ____________ }}
    {{ Q5: A[k] <= A[k+1] for any 0 <= k < n-1 }}
}
```

# Invariant Reasoning – Review

**func** dup([]) := []

dup(L ⧺ [x]) := dup(L) ⧺ [x] ⧺ [x]

```
/**
* Duplicates each element of an array
* @param arr an array of numbers
* @returns dup(arr)
*/
function duplicate(arr : number[]) : number[] { .. }
```

(a) Given the above function specification and definition of *dup*, come up with an invariant that is a weakening of the postcondition and an exit condition.

(b) Implement the loop with the invariant above.