Part 1

a. Intro to React

- Quickly set up React application with `npx create-react-app`
- Run the application with `npm start` (this is an alias for `npm run start`)

```
import React from 'react'
import ReactDOM from 'react-dom'

const App = () => (
  <div>
    <p>Hello world</p>
  </div>
)

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

Components

- React is **Declarative** by using **Components** that represents what to be rendered to the screen
- Names must be **capitalized**
- All content that needs to be rendered on screen is usually defined as **React Components**
- The component is defined as **a JavaScript function** and returns **JSX**
  - A components must return **one root element** or **an array of elements**

```
// we can wrap the return values with empty tag to get rid of extra div-elements
return (
    // <> is a shorter format for <React.Fragment>
    <>
```

```
        <h1>Hello</h1>
        <h2>Hi</h2>
    </>
)
```

- We can render **dynamic content** inside of a component as well. `<p> { a + b } </p>`
    - Use `{ }` to embed JS code within JSX, the result of the evaluation will be rendered

JSX

- HTML-like code that's compiled into JavaScript by **Babel**
- `create-react-app` set up the compilation process automatically
- Every tag needs to be closed (e.g. use `<br />` and not `<br>`)

Multiple Components

- A component can be called with its name in a tag such as `<App />`
- Components can be called within components, even multiple times

💡 A color philosophy of React is to **build reusable components**

Props: passing data to components:

- props are to components what **arguments are to functions**

```
// receiving props
const Hello = (props) => {
    <p> {props.name} {props.age} </p>
}
// passing props
<Hello name="Maya" age={26 + 10} />
```

b. JavaScript

- The official name of JS is **ECMAScript**
- JS is constantly updated to get new features from newer versions. However, browsers do not support all of the newest features. So code using the newer syntax is **transpiled** to an older version.

💡 **Babel** is the most popular transpiling tool

- **Node.js** is a **JS runtime environment,** we need it for executing JS code outside of the browser

Variables:

- `const` and `let` for defining block scoped variables
- `var` declares function-scoped variables and **should not be used** for it causes confusion

```
const x = 1           // defines a constant, value cannot be changed
let y = 5             // defines a normal variable, provides block-scoping

console.log(x, y)     // 1, 5
y += 10
console.log(x, y)     // 1, 15
y = 'sometext'        // variable type can change
console.log(x, y)     // 1, sometext
x = 4                 // error, because x is const
```

Arrays:

- `push` adds new element to an array, `concat` returns an array that concatenated with a new element

- React prefers the use of **immutable data structures,** so `concat` is more preferred than `push`
- use `forEach` to do something for each element in an array
- `map` creates a new array based on the old array and a function
- **Destructuring** `...` can unpack values from arrays and objects

```
const t = [1, -1, 3]
// contents of array can be modified even as a const
// the pointer to the array is constant but the array can change
t.push(5)

const t2 = t.concat(5) // creates new array [1, -1, 3, 5, 5]

console.log(t.length)  // 4
console.log(t[1])      // -1

// iterate through the array with for loop
t.forEach(value => {
  console.log(value)   // 1, -1, 3, 5 each to own line
}

// map
const t = [1,2,3]
const mt  = t.map(value => value * 2)
console.log(m1)  // [2, 4, 6]

// destructuring assignment: assign items of an array easily to variables
const t = [1, 2, 3, 4, 5]
const [first, second ...rest] = t

console.log(first, second)   // 1, 2
console.log(rest)   // [3, 4, 5]
```

Objects:

- Define objects with **object literals** by listing properties within `{ }`
- Reference or add properties with `.` or `[]`

```
// define objects with object literals
// values of the properties can be any type
const object1 = {
  name: "John Doe",
  age: 35,
  education: "PhD",
  grades: [2, 3, 5, 3],
}

// reference the properties using "dot" notation or brackets
console.log(object1.name)  // John Doe
console.log(object1[name]) // John Doe

// add properties using [] or .
object1['secret number'] = 1234
object1.address = "Waterloo"
```

Functions:

- The newest way is to use **arrow functions** (like lambda functions in Python)
- Before we have to use the `function` keyword (named functions or function expressions)

```
// arrow functions
const sum = (p1, p2) => {
  console.log(p1)
  console.log(p2)
  return p1 + p2
}

const result = sum(1, 5)
console.log(result)

// exclude parentheses if there is only a single parameter
const square = p => {
  console.log(p)
  return p * p
}

// when the function only contains a single expression, braces and `return` are not needed
const square = p => p * p

//--------BEFORE ARRAY FUNCTIONS--------------------
// named functions
function product(a, b) {
  return a * b;
}
// function expression
const average = function(a, b) {
  return (a + b) / 2
}
```

c. Component State, event handlers

Component helper functions

- we can define functions inside component that can access all props that are passed to the component

Destructuring

- a useful feature to destructure values from objects and arrays upon **assignment**
- `const { name, age } = props`
- `const Hello = ({ name, age }) => {}`

Page re-rendering:

- repeatedly calls a function with delay with `setInterval`
- repeatedly calling ReactDOM.render is not a good way to re-render components

Stateful component

💡 Before the state hook, we need to use class components to handle states

- We can add state that **could change during the lifecycle of the component** with **React's** `state hook`
- state hook
- `import { useState } from "react"`
- useState **returns a stateful value, and a function to update it**
    - `const [ counter, setCounter ] = useState(0)`
    - `counter` is assigned the initial value of state 0
    - `setCounter` is assigned to a function that will be used to modify the state
- When state modifying function is called, **React re-renders the component**, and the function body of the component gets **re-executed**

💡 useState creates the application state, and calling the function that changes the state causes the component to rerender

Event handling

- Registered to be called when specific events occur (e.g. clicking of a button)

```
<button onClick={() => setCounter(counter+1)}> plus </button>
```

NOTE: **Event handler is a function!**

- An event handler is supposed to be either a **function** or **a function reference**

Passing state to child components:

- write React components that are small and reusable across the application and even across projects.

d. A more complex state, debugging React apps

complex state:

- we can have multiple states by using the `useState` function multiple times

```
const [left, setLeft] = useState(0)
const [right, setRight] = useState(0)
```

- Alternatively, we can group the states together

```
const [clicks, setClicks] = useState({left: 0, right:0})
```

- And event handlers like this, using the **object spread syntax ...**

```
const handleLeftClick = () => {
  const newClick = {
    // spread operator
    ...clicks,
    left: clicks.left + 1
  }
  setClicks(newClicks)
}
```

- **NOTES! we must change state using the state changing function, or else it won't cause rerendering**

Conditional rendering:

- return different JSX based on conditionals (e.g. if statements)
- If you want to add conditionals in JSX, use the ternary operator (? :), regular if statements don't work in JSX

Debugging React applications

- First rule of web dev: keep the browser's developer console open at all times
- when we use `console.log`, don't combine objects by using the plus operator, instead use `,`

```
console.log("props value is", props)
```

- write the command `debugger` in your code, the execution will pause after it execute the debugger command
- the **React developer tools** extension is useful to inspect React components
- VSCode has snippet built in, type `log` and hit tab to autocomplete to `console.log()`

Rules of Hooks

- must not be called from inside of a loop, a conditional expression, or any place that's not a function defining a component. (To make sure the hooks are always called in the same order)

💡 hooks may only be called from the inside of a **React functional component**

Event Handling Revisited

- we can use function that returns a function as event handler

```
const hello = (who) => () => {
  console.log('hello', who)
}


<button onClick={hello('world')}>button</button>
```

- This can be utilized in defining generic functionality that can be customized with parameters

💡 **Do not define components inside components:** provides no benefits and leads to problems

What is the Virtual DOM?

- The DOM and why rendering it stinks
- The Virtual DOM
- The Virtual DOM as implemented by React
- Diffing / Reconciliation
- Peek at the source code

DOM(Document Object Model) : an API that represents an HTML page as a **tree structure** (each node is an HTML element)

- The purpose of the DOM is to allow JS to manipulate HTML (document.getElementById)
- The DOM updates is very inefficient (it re-renders every node)

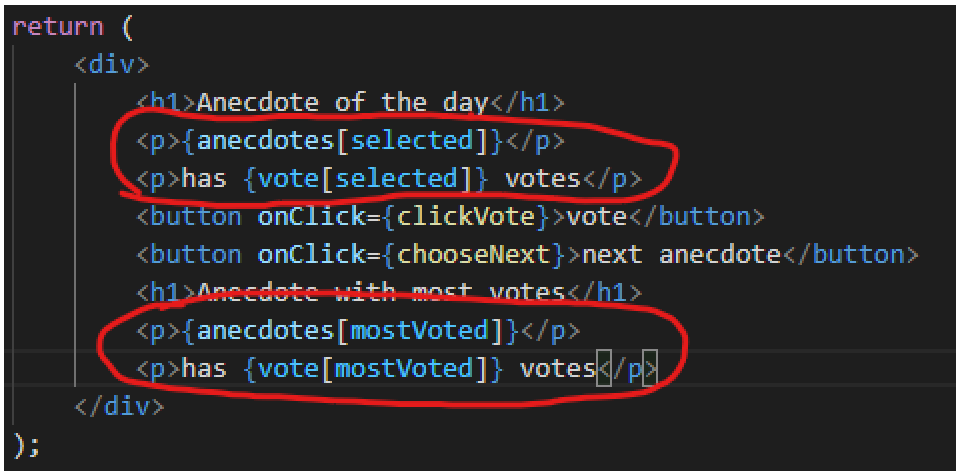
Virtual DOM:

- We render changes to the Virtual DOM before the real DOM
- Virtual DOM batch changes together for efficiency, updating is efficient
- It's a lightweight representation of the DOM
- It uses "Reconciliation" (diffing) to find the minimum # of modifications required for updating the DOM
- the key we uses allows React to differentiate elements on the same level

React Fiber: a rewrite of React's reconciliation algorithm

Takeaways From Exercises

- When we see repeated JSX elements, it might be a good idea to refactor them into another components
    - For example, the circled code in below image could be refactored



```
return (
    <div>
        <h1>Anecdote of the day</h1>
        <p>{anecdotes[selected]}</p>
        <p>has {vote[selected]} votes</p>
        <button onClick={clickVote}>vote</button>
        <button onClick={chooseNext}>next anecdote</button>
        <h1>Anecdote with most votes</h1>
        <p>{anecdotes[mostVoted]}</p>
        <p>has {vote[mostVoted]} votes</p>
    </div>
);
```

- It's better to destructure the props if possible
    - For example, if we know a component **Part** will accepts props of **name** and **exercise**, instead of declaring the components like

```
const Part = (props) => {
    return (
        <p>
            {props.name} {props.exercise}
        </p>
    );
};
```

    we can destructure the props like

```
const Part = ( {name, exercise} ) => {
    return (
        <p>
            {name} {exercise}
        </p>
    );
};
```