Intro to React
Lab 2
Intermediate

1 Online References

1.1 Basic references

https://www.w3schools.com/REACT/DEFAULT.ASP

1.2 Intermediate references

https://ibaslogic.com/react-tutorial-for-beginners

https://www.geeksforgeeks.org/reactjs-components/

https://scrimba.com/learn/learnreact

1.3 Official references

https://react.dev/learn

2 Lab Setup

We will continue working with any of the React projects that you had created in the previous lab.

As usual, if you wish to have the development server for multiple React apps running at the same time, you have to ensure that they are running on a different port. The default port is 3000, so you may use any other free port on your local machine (for e.g. 5000).

```
SET PORT=5000

npm start
```

The newly generated app should provide the same default view as previously.

3 Handling events

Event handlers are user defined functions added to JSX that will be called (triggered) in response to user interactions like clicking, hovering, focusing on form inputs, typing into a form field and so on.

Event handlers are:

a) are typically defined separately inside a component.
b) after being defined, they can then be passed to a child (or built-in) component's JSX tag as a prop.
c) have names that start with `handle`, followed by the name of the event (i.e. `handleClick`, `handleInput`, `handleMouseEnter`, etc).
d) have access to the component props

Project: `first-app`

Demonstrating a basic UI element (button) with no event handler
Folder to use: `events`
File to use: `App-v1.js`

Demonstrating a basic UI element (button) with a simple event handler passed to the `onClick` props
Folder to use: `events`
File to use: `App-v2.js`

List of all props for built-in components that take event handlers start with the keyword `on`
https://react.dev/reference/react-dom/components/common#reference

3.1 Event handler defined as arrow function in JSX

You can define an event handler inline in the JSX itself, typically through an arrow function (if the event handler logic is a single statement or is short).

Demonstrating a basic UI element (button) with an event handler in the JSX using arrow function format
Folder to use: `events`
File to use: `App-v3.js`

Demonstrating a custom component with an event handler that accesses its props
Folder to use: `events`
File to use: `App-v4.js`

3.2 Customized names for event handler props for child components

A typical design pattern is for built-in components (such as buttons) to contain styling, and their behavior is specified through event handlers passed down as props to them from a parent component. The props that receive event handlers for built-in components such as `<button>, <input>`, etc are typically predefined: for e.g. `onClick()`. However, for user defined components, you can name the props which you pass event handlers to them in any way you want.

Demonstrating a custom component with specialized props to pass event handlers to
Folder to use: `events`
File to use: `App-v5.js`

3.3 Accessing properties for React event object

All event handlers will by default receive an event object as a single parameter in its function definition. You can optionally choose to extract relevant properties (depending on the handler function itself, for e.g. MouseEvent, PointerEvent, FocusEvent, InputEvent, etc) and also implement some standard methods (such as preventDefault() and stopPropagation()).
https://react.dev/reference/react-dom/components/common#react-event-object

Make sure you have Developer tools Console viewable to see the console output from the app below.

Demonstrating accessing properties from an event object
Folder to use: `events`
File to use:
`App-v6.js`
`App-v6.css`

Note that the onChange props is a standard HTML attribute for various HTML form-related elements (e.g. <input>, <select>)
https://www.w3schools.com/tags/att_onchange.asp

https://www.w3schools.com/tags/ev_onchange.asp

4 Managing state with useState hook

Managing of state is critical in providing interactivity in a React app, because the UI will change in respond to user input and this input will typically be tracked using the state of a component.

State management cannot be handled using normal variables alone. This is because:

a) Local variables are not persisted between renders. Every time a component is re-rendered, React ignores changes to local variables
b) Changing local variables will not trigger a render. This however is necessary if the value of that variable is influencing the output being rendered.

The term render above simply refers to the process of calling a component.

Project: `first-app`

A basic but incorrect attempt to maintain state in a component via normal variables
Folder to use: `state`
File to use: `App-v1.js`

To update a component with new data, two things need to happen:
- Retain changes in variable values between renders.
- Trigger React to render (call) the component with new data (re-rendering).

The useState Hook provides those two things:
- A state variable to retain the data between renders.
- A state setter function to update the variable and trigger React to render the component again. This is typically called in the event handler.

Hooks are special functions that start with `use` which are only available while React is performing a render. They let you "hook into" different React features, the most important being state.

Hooks can only be called at the top level of your components. This is important because hooks rely on a stable call order on every render of the same component in order to ensure the correct state variable is returned in the event there is more than one state variable being used.

We use array destructuring to obtain a state variable and its corresponding setter function from the useState hook. The only argument for useState is the initial value of your state variable (which can be of different types) for the initial render.

This state variable registered with the useState hook is stored and tracked by React internally independent of the component. Every call to useState for a subsequent re-render after the initial render will return the latest value of the state variable.

This setter function must subsequently be used every time the state variable needs to be updated (typically in an event handler). Calling the setter function triggers a re-render of the component it is contained within.

The simplest use of the setter function is to pass it the new value that we want to set the state variable to, usually via an expression that involves the state variable directly (which allow us to access its current value)

A basic demo of the useState hook to maintain state correctly in a component
Folder to use: `state`
File to use: `App-v2.js`

Incorporating several state variables of different types in a component
Folder to use: `state`
File to use: `App-v3.js`

Notice that the `useState` Hook as well as the state variables are viewable in the React Developer Tools Component Panel. Try experimenting and changing the contents of the state variables to see the result in real time. The Component Panel provides a useful way for you to quickly check out new changes to your app quickly by modifying the relevant state variable values.

Using state variable changes to make dynamic changes to the DOM
Folder to use: `state`
Create an `images` subfolder in the `src` folder of the current React project. Copy the 3 jpg files (`cabbage, cereal, muffins`) into this `images` subfolder
File to use: `App-v3-1.js`
File to use: `App-v3.css`

Here, we can use the changes in the state variable to cause dynamic changes in the DOM, which is the core principle of how we add interactivity to a static webpage using React (or any other JavaScript based framework)

4.1 Isolation of state and lifting of state

State is local to a component. If a component is used more than once in JSX, each component instance has its own isolated state. Parent components cannot access their child component state.

Demonstrating that state variables of each component instance is isolated from each other
Folder to use: `state`
File to use:
`App-v4.js`
`App-v4.css`

Sometimes, you want the state of two different components to always change together in a synchronized manner. To do this, you can do what is known as "lifting state":

a) Remove relevant state variables from the child components.
b) Add these state variables to the common parent of both child components, together with event handlers that change these state variables
c) Pass these state variables and the event handlers as props to the child components

For each unique piece of state, there will be a component that owns it and is responsible for changing it before it is propagated down the component hierarchy. This principle is also known as having a “single source of truth”.

Demo basic lifting of state variables from child component to parent component
Folder to use: `state`
File to use:
`App-v5.js`

Another basic demo of parent component holding state variables that are passed to child component to synchronize the change in their display
Folder to use: `state`
File to use:
`App-v6.js`

Again, notice that the React Developer Tools Component Panel allows you to see the props passed down to child components from state variables in their parent components. As usual, you can try changing the values here to see the effect in real time.

4.2 Using state updater function to work with multiple state updates

When a state setter function is called to change a state variable (typically within an event handler), it triggers a re-render of the component (calls the component again) that the state variable is associated with. However, the actual re-render will only occur when the event handler has completed execution.

React keeps the state values “fixed” within the event handler and will only process all the state updates (in the event there are multiple calls to the setter functions) once all the code related to that event handler is complete. It will evaluate all the setter function calls consecutively one after the other, and then update the state variable with its new value. Then it will re-render (call the component again) using this updated state variable value, resulting in a consequent change in the DOM and the corresponding browser view.

Thus, a state variable’s value never changes within a given render, even when the setter function for the state variable is called numerous times within the event handler or even if the event handler code is asynchronous.

Demo multiple state updates in a single event handler
Folder to use: `state`
File to use:
`App-v7.js`

Sometimes, you may wish that the state variable be updated multiple times within the same event handler before the next render. In that case, you would use a state updater function which is passed as the sole argument to a state setter function in the form of an arrow function.
In this situation, the state setter function calls are still executed consecutively one after the other, however this time the results returned from the first state updater function be passed to the next state updated function and so on, resulting in the state variable being updated in the event handler itself.

It's common to name the updater function argument by the first letters of the corresponding state variable

Demo use of state updater function in a single event handler
Folder to use: `state`
File to use:
`App-v8.js`

4.3 Updating object state variables

Sometimes, if you have many individual primitive variables (string, number, boolean) for keeping state, you may want to combine them into a single object to make them easier to manage.

When you mutate a state object (by changing its properties) directly without calling the state setter function, React has no idea that the object has actually changed and will not trigger a render - so nothing happens. To actually trigger a re-render, you must create a new object with the same structure of the original object (but with new property values representing the updated state), and then pass it to the state setter function.

Demo 3 different ways to update object state variable
Folder to use: `state`
File to use: `App-v9.js`

4.4 Updating array state variables

You can also have an array as a state variable. Just like with objects, when you want to update an array stored in state, you need to create a new one (or make a copy of an existing one), and then set state to use the new array. You should avoid mutating / changing the existing array.

When dealing with arrays inside React state, you will need to avoid the methods in the left column, and instead prefer the methods in the right column:

Action	**Avoid (mutates existing array)**	**Prefer (creates a new array)**
Adding	`push, unshift`	`concat, [...arr] spread syntax`
Removing	`pop, shift, splice`	`filter, slice`
Replacing	`splice, arr[i] = ... assignment`	`map`
Sorting	`reverse, sort`	copy the array first

Demo updating the addition of items to an array state variable
Folder to use: `state`
File to use: `App-v10.js`

Notice that for the creation of the list of heroes, we are following the same concept that we saw earlier on rendering lists in the props chapter. We need to have a counter variable to provide the value for a `id` property that is subsequently used as a unique value for the `key` props in all the `<li>` items.

Demo updating the deletion of items from an array state variable

Folder to use: `state`
File to use: `App-v11.js`

For deleting items, we simply use the filter method to create a new array with the item to be deleted filtered out.

Demo updating the replacement of items within an array state variable
Folder to use: `state`
File to use: `App-v12.js`

For replacing items, we can create a copy of the original array and then locate the item we are looking for and replace it with a new item.

Demo inserting new items before an existing item in an array state variable
Folder to use: `state`
File to use: `App-v13.js`

To insert new items, we first identify the location (index) in the array where we want to insert the new item and then use the slice method to perform the insertion.

Demo sorting items in an existing array state variable
Folder to use: `state`
File to use: `App-v14.js`

To sort items in an array, we identify a suitable comparison arrow function to be passed to the sort method to determine the sorting order of the items in the array.

4.5 Choosing state structure

So far, we have seen in previous labs that our state variables can be primitive types (string, number, boolean) or complex types (objects or arrays). The choice of the kind of state variables you use in your app and how you work with them can influence how easy it is to work with your code when your app grows in complexity.

Some underlying principles on how to choose the right state structure:

https://react.dev/learn/choosing-the-state-structure

5 Working with forms

Forms are a very important aspect of any web app because they are the primary way through which a user will enter information that the app requires during the process of logging in, signing up, making purchases, selecting options, etc. Learning to create effective and user-friendly forms is essential for developers looking to build engaging and interactive web applications. React provides several features and techniques for creating and managing forms based on the various aspects of state management and event handling as well as performing form validation.

There are two approaches to handling forms: controlled and uncontrolled components

a) Controlled components is the most common approach, and here the form data is handed through the use of useState (and other related) hooks. When a component is controlled, the value of form elements is stored in a state, and any changes made to the value are immediately reflected in the state
b) Uncontrolled Components is where form data is handled by the DOM rather than by React. The DOM maintains the state of form data and updates it based on user input.

5.1 Basic form elements (controlled components)

Project: `first-app`

Folder to use: `forms`
File to use: `App-v1.js`

In this example, we create state variables to keep track of the values of the all various form elements. We also pass the relevant event handlers to the `onChange` props of these elements and access the `target.value` property from the event object

Note that the `for` attribute that is typically found in the `label` elements is rewritten as `htmlFor` instead, since in JSX. This is because attributes written in JSX become keys of JavaScript objects. JavaScript has limitations on key / variable names: e.g. cannot have dashes, cannot be reserved words (`for` is reserved keyword).

5.2 Implementing multiple checkboxes

In the previous lab, we used separate state variables to keep track of the status (checked / unchecked) of each of the check boxes in the form. This approach is fine when the number of check boxes is small, but for a large number of check boxes, it becomes redundant and impractical.

For that situation, we can use two arrays as state variables:

- An array that keeps track to the info related to each of the checkboxes (labels, etc) - this will be an array of strings / objects
- An array that keeps track of the status of each checkbox (either ticked or unticked) - this will be an array of Booleans which will be maintained as a state variable

Folder to use: `forms`
File to use:
`App-v2.js`
`App-v2.css`

As you interact with the app, you can use the React Developer Tools extension to check the state variable array that represents the status of each checkbox as well as the state variable that represents the total of the prices associated with all the items whose checkboxes are checked.

5.3 Handling form submission for multiple form fields

When there are multiple form elements (e.g. text field, email field, text area, drop down lists, check boxes, radio buttons, etc) within a single large top-level <form> parent element, using separate state variables to track each one of them is not practical or scalable. In that case, it is better to use a single object whose properties will hold the contents of all these nested child form elements and update the corresponding property in a single event handler that is called whenever any of the form elements are updated.

We also optionally use the `event.preventDefault()` to prevent the default behavior of the browser when a submit form occurs (this would be to refresh the forms fields).

Typically, when form data is submitted, the React app will send it in a HTTP request to a backend service (which could be implemented using a suitable backend framework such as Node.js/Express.js, Laravel, etc). This service implements the business logic to process the form data properly and store it into a final backend database. This is not demonstrated here because the workshop labs here are only focusing on front-end client-side development using React.

Folder to use: `forms`
File to use:
`App-v3.js`

As you interact with the app, you can use the React Developer Tools extension to check the object state variable and how the property values within it change in association with an update to the various child form elements (text field, email field and text area).

5.4 Form validation logic

Validating forms refers to the process of checking user input in form elements to ensure that the input meets specific criteria before it is actually processed. Common requirements include:

- Ensuring a form field has data within it (input is mandatory, blank or missing input is considered invalid)
- Specific data (for e.g. telephone number, date, email address, etc) must follow a predefined format
- Passwords must contain certain characters (e.g. one uppercase letter, one symbol, and a number) and must be of a certain length
- Numbers entered must be within a certain range
- Strings entered must be of certain length

Form validation can be done either client-side in the browser (by vanilla JavaScript or a JavaScript framework such as React or Angular) or server-side (by the backend service that receives the data in a HTTP request from the browser).

HTML5 already provides some attributes that can be used with certain form elements to perform validation, independently of JavaScript itself:
https://www.w3schools.com/html/html_form_attributes.asp
Examples of this include:

- Making it mandatory for a field to have data using `required`
- Constraining the length of data, for e.g. `minlength, maxlength` for text data and `min` and `max` for numeric data
- Restricting the type of data using type `(<input type="email" ….>)`
- Specifying data patterns using `pattern` and a regular expression

In addition to this, we can create a function within a React app to implement custom validation functionality of our own. The advantage of doing this compared to a using HTML5 validation

HTML5 validation only provides error messages when the entire form is submitted. With React, you can validate the content of a form element while content is being dynamically entered into it.
HTML5 validation provides standard error messages, regardless of the form context (i.e. e-commerce purchase form, login form, account signup form), whereas React allows you to customize the error messages in anyway you want.

You can mix both HTML5 validation and React-based validation together in a single form if you wish (as shown in the example below), or you can choose to use one approach singularly.

Folder to use: `forms`
File to use:
`App-v4.js`

The First Name, Last Name, Age and Email fields are controlled by HTML5 validation. Play around with trying to enter invalid values into these fields. With the exception of the Age field, you will be able to do so, and this invalid value is only detected when an attempt is made to submit the form, to which a common error message is provided.

The Job and Year fields have their validation performed by logic inside the event handler that is triggered for change in any of the form fields. This allows a custom error message to be displayed immediately when the field content is invalid, and not only when the form is about to be submitted. We can also implement another validation check at form submission time if we wish.