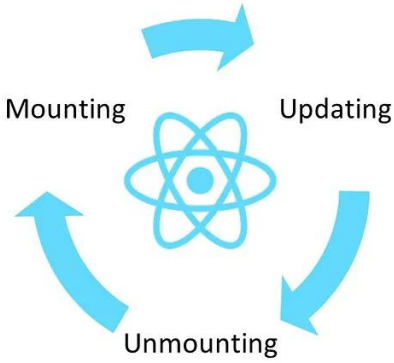
React Ref hook

Component Lifecycle

- Series of events that occur from a component's creation to its removal.

- The lifecycle is divided into three main phases:
  1. **Mounting** – Component is created and inserted into the DOM.
  2. **Updating** – Component is re-rendered due to changes in props or state.
  3. **Unmounting** – Component is removed from the DOM.

React Component Lifecycle



https://medium.com/@itsanuragjoshi/understanding-react-component-lifecycle-class-vs-functional-components-d1d6a974deda

Mounting Phase

- Methods involved during mounting:
  - **constructor()**: Initialises state and binds methods.
  - static **getDerivedStateFromProps()**: Syncs state with props before render.
  - **render()**: Returns JSX to display.
  - **componentDidMount()**: Invoked once after component mounts.

Updating Phase

- Methods involved during updating:
- static **getDerivedStateFromProps()**: Runs before every re-render.
- **shouldComponentUpdate()**: Determines if a re-render is needed.
- **render()** : Re-renders JSX.
- **getSnapshotBeforeUpdate()**: Captures DOM info before update.
- **componentDidUpdate()**: Invoked after updates are flushed to the DOM.

Unmounting Phase

- Unmounting happens when a component is removed from the DOM.
- **componentWillUnmount()** : Called right before the component is unmounted.
- Used for cleanup:
  - Removing event listeners
  - Canceling API requests
  - Clearing timers

React Hooks Equivalent

- In functional components, React Hooks manage lifecycle behaviour:

- **useEffect()** – Handles mounting, updating, and cleanup (unmounting).
- **useState()** – Manages local component state.

```
useEffect(() => {
   console.log('Component mounted');
   return () => console.log('Component unmounted');
}, []);
```

- Lifecycle methods provide control over component behavior.
- Class components use explicit lifecycle methods.
- Functional components use hooks like **useEffect**.

Imperative Programming

- Tell the computer **how** to do something — step by step instructions.
- Focuses on **control flow**: loops, conditions, and explicit DOM manipulation.
- Common in vanilla JS or jQuery

```
const list = document.getElementById("items");
const li = document.createElement("li");
li.textContent = "New item";
list.appendChild(li);
```

Characteristics of Imperative Code

- Direct DOM manipulation.
- Manually manage state and UI updates.
- Higher chance of bugs as complexity grows.
- Closer to how the computer “thinks” than how you “want it to look”.

What Is Declarative Programming

- You describe **what you want**, then React figures out **how** to do it.
- Focuses on **UI state**, not the steps to change it.

```
function TodoList({ items }) {
  return <ul>{items.map(item => <li key={item}>{item}</li>)}</ul>;
}
```

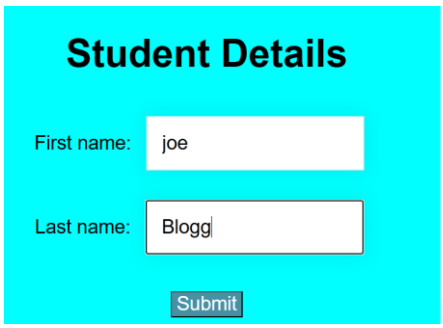
- React automatically updates the DOM when items changes.

- Declarative Programming in React
- UI = function of state
- React handles the DOM efficiently using the Virtual DOM.
- Cleaner, more predictable, and easier to debug.
- Encourages component reusability and unidirectional data flow.

React Refs

- **Refs** (short for references) allow you to directly access and interact with DOM elements in React.
- Unlike state variables, refs **do not trigger re-renders** when updated.
- Commonly used for:
  - Managing focus or text selection
  - Triggering animations
  - Reading input values without using state

Why refs




```
4  function LoginExample() {
5
6    const [enterFName, setFName] = useState('');
7    const [enterLName, setLName] = useState('');
8
9    function handleFNameInput(evt) {
10     const name = evt.target.value;
11
12     setFName(name);
13     console.log(enterFName);
14   }
15
16   function handleLNameInput(evt) {···
21   }
22
23   function handleSubmitForm(event) {
24     event.preventDefault();
25     // could send enteredEmail to a backend server
26   }
27   return (
28     <div>
29     <form className="enrolForm"  onSubmit={handleSubmitForm}>
30       <h1>Student Details</h1>
31       <label>First name:</label>
32       <input type="text" onChange={handleFNameInput} name="fname" />
33       <br />
34       <label>Last name:</label>
35       <input type="text" name="lname" onChange={handleLNameInput} />
36       <br />
37       <br />
38       <input type="submit" value="Submit" />
```

Creating and Using a Ref

- import **useRef** from **React**
- `import React, { useRef } from 'react';`

- Create a ref and attach it to an element:

- `const nameInputRef = useRef();`
- `<input ref={nameInputRef} type="text" />`

- Access the DOM node directly:

- `console.log(nameInputRef.current.value);`

- Every ref object created by **useRef()** has a **.current** property.
- **.current** holds a mutable reference to a DOM element or value.
- Unlike state, updating .current does not trigger a re-render.

```
const inputRef = useRef(null);
<input ref={inputRef} />;
console.log(inputRef.current.value); // direct DOM access
```

Refs vs State

	Refs	State
Causes re-render	No	Yes
Access DOM directly	Yes	No
Stores transient data	Yes	No
Best for	Focus, reading values	Displayed data

Benefits of Using Refs Here

- Avoids unnecessary state updates for simple read-only input retrieval.
- Simplifies event handling code.
- Great for quick form prototypes or when validation isn't required.
- Combine with **state** later for advanced validation or conditional UI rendering.

```jsx
import React, { useRef } from 'react';

function LoginExample() {
  const fNameRef = useRef();
  const lNameRef = useRef();

  function handleSubmitForm(event) {
    event.preventDefault();
    console.log(`First Name: ${fNameRef.current.value}`);
    console.log(`Last Name: ${lNameRef.current.value}`);
  }

  return (
    <form onSubmit={handleSubmitForm}>
      <h1>Student Details</h1>
      <label>First name:</label>
      <input ref={fNameRef} type="text" />
      <br />
      <label>Last name:</label>
      <input ref={lNameRef} type="text" />
      <br />
      <input type="submit" value="Submit" />
    </form>
  );
}
export default LoginExample;
```

When to use Imperative Code

- Focusing or scrolling elements (via **refs**).
- Third-party libraries that directly manipulate DOM.
- Performance-critical tasks.
- Caution: Keep it isolated: React should stay declarative overall.

- Refs (References) provide a way to directly access DOM elements or React components.
- Unlike state, changing a ref does not trigger a re-render.
- Created using useRef() in functional components.
- Refs allow you to interact directly with the DOM or components **imperatively,** while React's usual workflow is **declarative**.
- Use them sparingly