Methods for dealing with properties (Props) and setting default values

```jsx
//Direct destructuring in the function parameter
const Greetings = ({ name = "Guest" , age=20 }) => {
    return <h1>Hello, {name} ! your age {age}</h1>;
};

//Receiving the full props object and manually extracting values
const Greeting = (props) => {
    return <h1>Hello, {props.name? props.name:"Guest"} ! your age {props.age? props.age:20}</h1>;
};
```

Another example:

Direct destructuring in the function parameter

```jsx
const Counter = ({
  title = "Counter Example",
  initialCount = 100,
  backgroundColor = "lavender",
}) => {
  const [count, setCount] = useState(initialCount);
  const [bgColor, setBGColor] = useState(backgroundColor);
```

Receiving the full props object and manually extracting values

```jsx
const Counter = (props) => {
  //initilaize class properties
  const title = props.title ?? "Default Counter";
  const backcolor = props.backcolor ?? "#ddd";
  const numberOfClicks = props.numberOfClicks ?? 100;
  //initialize state
  const [count, setCount] = useState(numberOfClicks);
  const [bgColor, setBgColor] = useState(backcolor);
```

Conditional Rendering

Conditional Rendering means **showing or hiding UI elements based on certain conditions**.

Note :React doesn't have built-in if statements in JSX (if Statement outside JSX)

- **Ternary Operator** (condition ? truePart : falsePart)
- **Logical AND (&&)** : Used when you want to render something only if a condition is true (no else part):

```
return (
  <div>
    {props.showMessage && <p>This is a secret message.</p>}
  </div>
);
```

If props.showMessage is true, the paragraph is shown. Otherwise, nothing is rendered.

mount and unmount - useEffect

import { useState, useEffect } from 'react';

```
useEffect(() => {
  console.log("Counter mounted"); //componentDidMount
  setCount(numberOfClicks);
  setBgColor(backcolor);
  return () => {
    console.log("Counter unmounted"); //componentWillUnmount
  };
},[]);
```

At **mount** (the first appearance):

- It show `Counter mounted` and sets the number of clicks and background color.

At **unmount** (when it disappears):

- It show `Counter unmounted`

The **empty array** [ ] means that the useEffect:

- Will run only once at mount (initial render) .
- Will execute the cleanup method inside `return` during unmount.

If there is **no** [ ] , then any change will trigger both unmount and mount executions.

```
useEffect(() => {
  console.log("Count or color changed");
}, [count, color]); // ← Whenever count or color changes, the useEffect runs
```

Instead of using [ ] (which means the effect runs only once on mount), you can **put variables inside the array** so that useEffect runs every time one of those variables changes.

Custom Hooks

Definition of a Custom Hook

A **Custom Hook** in React is a **JavaScript function** that starts with the word "use" and allows you to **encapsulate and reuse logic** that involves React hooks (like useState, useEffect, useContext, etc.).

Custom Hooks **do not replace** built-in hooks; instead, they build on top of them to share logic across multiple components without duplicating code.

Use Cases for Custom Hooks

1. **Fetching Data**
   Example: useFetch(url) — handles API calls, loading, and errors.
2. **Form Handling**
   Example: useForm(initialValues) — manages form state, validation, and submission.
3. **Event Listeners or DOM Interaction**
   Example: useClickOutside(ref, handler) — detects clicks outside a given element.
4. **Authentication**
   Example: useAuth() — tracks user login status, roles, and permissions.

Creation of Custom Hooks → From Cheat Sheet

Two ways to get data from the API

Fetch then then

```
const App = () => {
  const [counterData, setCounterData] = useState([]);
  const API_URL = "https://njuneidi.github.io/calculator/data.json";
  const getData = () => {
    fetch(API_URL)
      .then((response) => response.json())
      .then((data) => {
        console.log(data);
        setCounterData(data);
      });
  };
```

async await

```
const fetchData = async () => {
  try {
    const response = await fetch("http://localhost:5173/data.json");
    // if (!response.ok) {
    //   throw new Error("HTTP error, status = " + response.status);
    // }
    const data = await response.json();
    console.log(data);
    setCounterData(data);
  } catch (error) {
    console.log(error);
  }
};
```

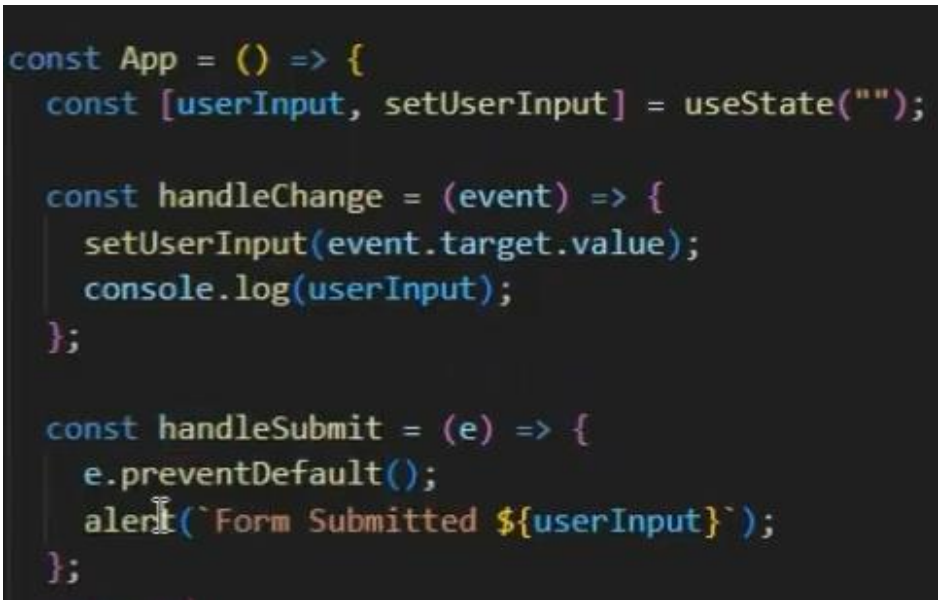
handleChange , handleSubmit (for one input)

The purpose of handleChange is :

- To instantly update the value of the userInput variable whenever the user types or modifies anything inside the textbox.
- Every time the user types a character or makes a change in the <input> , handleChange is automatically triggered. It captures the value entered by the user (event.target.value) and updates the state using setUserInput.

The purpose of handleSubmit is:

- To prevent page reload.
- In forms, the default behavior is for the page to reload upon clicking submit.
- By using e.preventDefault() , the reload is prevented.



```
const App = () => {
  const [userInput, setUserInput] = useState("");

  const handleChange = (event) => {
    setUserInput(event.target.value);
    console.log(userInput);
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Form Submitted ${userInput}`);
  };
```

```
  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="name">
        Enter Your Text
        <input
          type="text"
          name="name"
          id="name"
          value={userInput}
          onChange={handleChange}
        />
      </label>

      <button type="submit">Submit</button>
    </form>
  );
};
export default App;
```

handleChange , handleSubmit (for more than one input)

No Destructuring ; accesses name and value directly from e.target

```
import { useState } from "react";

function ContactForm() {
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    suppport: "",
    message: "",
  });
  const handleChange = (e) => {
    setFormData({
      ...formData, //  { name: "", email: "", suppport: "", message: "",}
      [e.target.name]: e.target.value, // email: "Ahmad@t.com
    });
  };
```

or : Destructuring is used to extract name and value from event.target

```
  const handleChange = (event) => {
    const { name, value } = event.target;
    console.log(formData);
    setFormData({ ...formData, [name]: value });
    // setFormData(event.target.value);
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Form Submitted ${formData.name} ${formData.email}`);
  };
```

```jsx
return (
  <form onSubmit={handleSubmit}>
    <label htmlFor="name">
      Enter Your Text
      <input
        type="text"
        name="name"
        id="name"
        value={formData.name}
        onChange={handleChange}
      />
    </label>
```

```jsx
    <label htmlFor="email">
      Enter Your Email
      <input
        type="email"
        name="email"
        id="email"
        value={formData.email}
        onChange={handleChange}
      />
    </label>

    <button type="submit">Submit</button>
    <button
      type="reset"
      onClick={() => {
        setFormData({ name: "", email: "" });
      }}
    >
      Reset
    </button>
  </form>
```

Validation Form

```
const handleSubmit = (e) => {
  e.preventDefault();
  if (!formData.name || !formData.email) {
    alert("Please fill data");
    return;
  }
  alert(`Form Submitted ${formData.name} ${formData.email}`);
};
```

return → Stop form submission

Another way using useState()

```
const [errors, setErrors] = useState({});
const navigate = useNavigate();

const ValidateForm = () => {
  const newError = {};
  if (!formData.name.trim()) {
    newError.name = "You should fill Name";
  }
  if (!formData.message.trim()) {
    newError.message = "You should fill Message";
  }
  if (!formData.email.match(/^\S+@\S+\.\S+/)) {
    newError.email = "Invalid email";
  }
  return newError;
};
```

```
const handleSubmit = (e) => {
  e.preventDefault();
  const validateErro = ValidateForm();
  if (Object.keys(validateErro).lengthe === 0) {
    console.log(`Form Sumitted`, formData);
    navigate("/thank-you");
  } else {
    setErrors(validateErro);
  }
};
```

To show the error to the user:

```
return (
  <div>
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="name">Enter Your Name</label>
        <input
          type="text"
          name="name"
          id="name"
          value={formData.name}
          onChange={handleChange}
        />
        {errors.name && <div style={{ color: "red" }}>{errors.name}</div>}
      </div>
```

React Router

React Router:

- **BrowserRouter:** Wraps the entire application
- **Routes/Route:** Defines URL paths and their components
- **Link:** Navigation between pages without full reload
- **useNavigate:** Programmatic navigation

Install React Router:

npm install react-router-dom

```jsx
import { BrowserRouter as Router, Routes, Route } from "react-router-dom
import { About, Contact, Home } from "./pages";

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />}></Route>
        <Route path="/about" element={<About />}></Route>
        <Route path="/contact" element={<Contact />}></Route>
      </Routes>
    </Router>
  );
};
export default App;
```

Note : If you do not specify a path for a Route, **it will match all URLs by default.** In this case, the component inside that Route will always render.

---

useNavigate

---

import { useNavigate } from "react-router-dom";
const navigate = useNavigate();

```jsx
const handleSubmit = (e) => {
  e.preventDefault();
  console.log("Form submitted", FormData);
  navigate("/thank-you");
};
```

useNavigate allows you to navigate programmatically inside a React application without having to click on a <Link> component (Instead of clicking a link to move to another page, you can trigger navigation through your code)

The difference between Link and Path (For quick reading only)

Link

- Link is a **component** used to navigate between pages inside a React application **without reloading** the entire page.
- Its main purpose is to **replace traditional <a> tags** in a way that maintains the smooth experience of a **Single Page Application (SPA)**.
- When you click a Link, it changes the URL and displays the corresponding page without a full page reload.

✅ Example usage:

```
import { Link } from 'react-router-dom';

<Link to="/about">Go to About Page</Link>
```

- Here, when the user clicks the Link, the URL changes to /about, and **React Router** displays the component associated with that path.

Path

- Path is simply a string that represents a **URL** route pattern that a **Route** matches.
- It is used to define when a certain component should be displayed.
- Path is more related to the Route setup, not directly to the user interaction.

✅ Example usage:

```
import { Route } from 'react-router-dom';

<Route path="/about" element={<AboutPage />} />
```

- Here, path="/about" means that when the URL is /about, the AboutPage component will be rendered.

Summary of the difference:

Item	Purpose	Example
`Link`	Navigation within the app	`<Link to="/about">About</Link>`
`Path`	Defines which route to match	`<Route path="/about" element={<AboutPage />} />`