**REACT day 1**

**Source: REACT.DEV**

-React is a JavaScript library for rendering user interfaces (UI). UI is built from small units like buttons, text, and images. React lets you combine them into reusable, nestable components. From web sites to phone apps, everything on the screen can be broken down into components. In this chapter, you’ll learn to create, customize, and conditionally display React components.

**In this chapter**

* [How to write your first React component](https://react.dev/learn/your-first-component)
* [When and how to create multi-component files](https://react.dev/learn/importing-and-exporting-components)
* [How to add markup to JavaScript with JSX](https://react.dev/learn/writing-markup-with-jsx)
* [How to use curly braces with JSX to access JavaScript functionality from your components](https://react.dev/learn/javascript-in-jsx-with-curly-braces)
* [How to configure components with props](https://react.dev/learn/passing-props-to-a-component)
* [How to conditionally render components](https://react.dev/learn/conditional-rendering)
* [How to render multiple components at a time](https://react.dev/learn/rendering-lists)
* [How to avoid confusing bugs by keeping components pure](https://react.dev/learn/keeping-components-pure)
* [Why understanding your UI as trees is useful](https://react.dev/learn/understanding-your-ui-as-a-tree)

How to open project straight from folder, go up to location tab, write cmd, enter , after reaching cmd, type code .

**a React component is a JavaScript function that you can sprinkle with markup.**

**That’s why we use jsx files for react comonents cause it allows us to add html(markup) also to our js files.**

**NOW talking about components:**

**How to build your 1st component?:**

And here’s how to build a component:

**Step 1: Export the component**

The export default prefix is a standard JavaScript syntax (not specific to React). It lets you mark the main function in a file so that you can later import it from other files. (More on importing in the topic under Importing and Exporting Components!)

**Step 2: Define the function**

With function Profile() { } you define a JavaScript function with the name Profile.

**Pitfall**

React components are regular JavaScript functions, but their names must start with a capital letter or they won’t work!

**Step 3: Add markup**

The component **returns** an **EG: <**img /> tag with src and alt attributes. <img /> is written like HTML, but it is actually JavaScript under the hood! This syntax is called JSX, and it lets you embed markup inside JavaScript.

Return statements can be written all on one line, as in this component:

return <img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />;

But if your markup isn’t all on the same line as the return keyword, you must wrap it in a pair of parentheses:

return (

<div>

<img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />

</div>

);

**Pitfall**

Without parentheses, any code on the lines after return will be ignored!

Now that you’ve defined your Profile component, you can nest it inside other components. For example, you can export a Gallery component that uses multiple Profile components:

function **Profile()** {

**return (**

**<img**

**src="https://i.imgur.com/MK3eW3As.jpg"**

**alt="Katherine Johnson"**

**/>**

**);**

}

**export** default function **Gallery()** {

**return** (

<section>

<h1>Amazing scientists</h1>

**<Profile />**

**<Profile />**

**<Profile />**

</section>

);

}

Notice the difference in casing:

* <section> is lowercase, so React knows we refer to an HTML tag.
* <Profile /> starts with a capital P, so React knows that we want to use our component called Profile.

And Profile contains even more HTML: <img />. In the end, this is what the browser sees:

<section>

<h1>Amazing scientists</h1>

<img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />

<img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />

<img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />

Yo img haru chai Profile component le return garirathyo ni, aba browser le esari dekhirhuncha

</section>

NOT:

<section>

<h1>Amazing scientists</h1>

**<Profile />**

**<Profile />**

**<Profile />**

</section>

**Nesting and organizing components**

Components are regular JavaScript functions, so you can keep multiple components in the same file. This is convenient when components are relatively small or tightly related to each other. If this file gets crowded, you can always move Profile to a separate file. You will learn how to do this shortly on the [page about imports.](https://react.dev/learn/importing-and-exporting-components)

Because the Profile components are rendered inside Gallery—even several times!—we can say that Gallery is a **parent component,** rendering each Profile as a “child”. This is part of the magic of React: you can define a component once, and then use it in as many places and as many times as you like.

you can compose, order and nest components to design whole pages. For example, the documentation page you’re reading is made out of React components:

<PageLayout>

<NavigationHeader>

<SearchBar />

<Link to="/docs">Docs</Link>

</NavigationHeader>

<Sidebar />

<PageContent>

<TableOfContents />

<DocumentationText />

</PageContent>

</PageLayout>

In this example, the **to** prop of the **Link** component is set to "/docs," which is a relative path. (You know ni application banauda, jaile relative path dinuparcha because the client is definitely not gonna use your computer right, jun sukai language use garepani….tara different languages ma relative paths dine way/syntax different huncha. ).Make sure you have a route or endpoint set up in your React application to handle this path if you want it to navigate to a specific page or component.

As your project grows, you will notice that many of your designs can be composed by reusing components you already wrote, speeding up your development. Our table of contents above could be added to any screen with <TableOfContents />! You can even jumpstart your project with the thousands of components shared by the React open source community like [Chakra UI](https://chakra-ui.com/) and [Material UI.](https://material-ui.com/)

Components can render other components, but **you must never nest their definitions:**

export default function Gallery() {

// 🔴 Never define a component inside another component!

function Profile() {

// ...

}

// ...

}

The snippet above is [very slow and causes bugs.](https://react.dev/learn/preserving-and-resetting-state#different-components-at-the-same-position-reset-state) Instead, define every component at the top level:

export default function Gallery() {

// ...

}

// ✅ Declare components at the top level

function Profile() {

// ...

}

When a child component needs some data from a parent, [pass it by props](https://react.dev/learn/passing-props-to-a-component) instead of nesting definitions.

**Recap**

You’ve just gotten your first taste of React! Let’s recap some key points.

* React lets you create components, **reusable UI elements for your app.**
* In a React app, every piece of UI is a component.
* React components are regular JavaScript functions except:
  1. Their names always begin with a capital letter.
  2. They return JSX markup.

**Note:**

As your project grows, you will notice that many of your designs can be composed by reusing components you already wrote, speeding up your development. Our table of contents above could be added to any screen with <TableOfContents />! You can even jumpstart your project with the thousands of components shared by the React open source community like Chakra UI and Material UI.

**DIVING deep into components:**

Your React application begins at a “root” component. Usually, it is created automatically when you start a new project. For example, if you use [CodeSandbox](https://codesandbox.io/) or if you use the framework [Next.js](https://nextjs.org/), the root component is defined in pages/index.js. In these examples, you’ve been exporting root components.

What is CODESANDBOX?

WHAT IS CODE SANDBOX?

CodeSandbox is an online integrated development environment (IDE) for web development. It allows developers to quickly create, share, and collaborate on web projects directly in the browser. CodeSandbox supports various web development technologies, including HTML, CSS, JavaScript, React, Vue, and more.

Key features of CodeSandbox include:

1. **Instant Setup:** CodeSandbox provides a quick and hassle-free way to start coding without the need for local development environments. You can start coding right in your web browser.
2. **Collaboration:** Multiple users can collaborate on a single project in real-time. This is particularly useful for pair programming or working on group projects.
3. **Frameworks and Libraries:** CodeSandbox supports a variety of front-end libraries and frameworks, such as React, Vue, Angular, and more. You can choose the technology stack that suits your project.
4. **Live Preview:** You can see the live preview of your code as you type, making it easy to iterate and test your changes without the need for manual refreshes.
5. **Sharing:** CodeSandbox allows you to easily share your projects by generating a shareable link. This makes it convenient for showcasing your work or seeking help from others.
6. **Templates and Examples:** CodeSandbox provides templates and examples for various frameworks and libraries, helping you kickstart your projects with pre-configured setups.
7. **GitHub Integration:** You can import projects from GitHub or export your projects to GitHub directly from CodeSandbox, making it seamless to integrate with version control.

CodeSandbox is widely used by developers for prototyping, learning, and collaborating on web projects. It provides a user-friendly environment, making it accessible to both beginners and experienced developers. You can access CodeSandbox at [codesandbox.io](https://codesandbox.io/).

odeSandbox is an online integrated development environment (IDE) for web development. It allows developers to quickly create, share, and collaborate on web projects directly in the browser. CodeSandbox supports various web development technologies, including HTML, CSS, JavaScript, React, Vue, and more.

Key features of CodeSandbox include:

1. **Instant Setup:** CodeSandbox provides a quick and hassle-free way to start coding without the need for local development environments. You can start coding right in your web browser.
2. **Collaboration:** Multiple users can collaborate on a single project in real-time. This is particularly useful for pair programming or working on group projects.
3. **Frameworks and Libraries:** CodeSandbox supports a variety of front-end libraries and frameworks, such as React, Vue, Angular, and more. You can choose the technology stack that suits your project.
4. **Live Preview:** You can see the live preview of your code as you type, making it easy to iterate and test your changes without the need for manual refreshes.
5. **Sharing:** CodeSandbox allows you to easily share your projects by generating a shareable link. This makes it convenient for showcasing your work or seeking help from others.
6. **Templates and Examples:** CodeSandbox provides templates and examples for various frameworks and libraries, helping you kickstart your projects with pre-configured setups.
7. **GitHub Integration:** You can import projects from GitHub or export your projects to GitHub directly from CodeSandbox, making it seamless to integrate with version control.

CodeSandbox is widely used by developers for prototyping, learning, and collaborating on web projects. It provides a user-friendly environment, making it accessible to both beginners and experienced developers. You can access CodeSandbox at [codesandbox.io](https://codesandbox.io/).

Most React apps use components all the way down. This means that you won’t only use components for reusable pieces like buttons, but also for larger pieces like sidebars, lists, and ultimately, complete pages! Components are a handy way to organize UI code and markup, even if some of them are only used once.

[React-based frameworks](https://react.dev/learn/start-a-new-react-project) (eg,: NEXT.js) take this a step further. Instead of using an empty HTML file and letting React “take over” managing the page with JavaScript, they *also* generate the HTML automatically from your React components. This allows your app to show some content before the JavaScript code loads.

Still, many websites only use React to [add interactivity to existing HTML pages.](https://react.dev/learn/add-react-to-an-existing-project#using-react-for-a-part-of-your-existing-page) They have many root components instead of a single one for the entire page. You can use as much—or as little—React as you need.

**Importing and Exporting Components**

The magic of components lies in their reusability: you can create components that are composed of other components. But as you nest more and more components, it often makes sense to start splitting them into different files. This lets you keep your files easy to scan and reuse components in more places.

……………………………………………………………………………………………….

Related components lai chai euttai file ma halne baru. Eg.: mathi hamile profile ra gallery bhanne component banayera teslai euutai file ma halyam because profile lai nai gallery bhitra render garirathyam ani export default Gallery garirathyam.

Eta export **default** Gallery means:

Euta jsx file ma multiple components ta huna sakcha ni technically(esto practice chai ramro oina), so default ma chai kun component export garne bhanera tyo euta jsx fie because export ta euta main component matra extract garna milcha ni, so aru components cha bhane ni tyo main(root) component bhitra render garera, tyo main component lai matra export garne ho.

…………………………………………………………………………………………………..

**NOW You will learn**

* What a root component file is
* How to import and export a component
* When to use default and named imports and exports
* How to import and export multiple components from one file
* How to split components into multiple files

These(Profile and Gallery; the components we made previously in App.js file) currently live in a **root component file,** named App.js in this example. Depending on your setup, your root component could be in another file, though. If you use a framework with file-based routing, such as Next.js, your root component will be different for every page.

Let’s understand this:

import Gallery from './Gallery.js';

here,

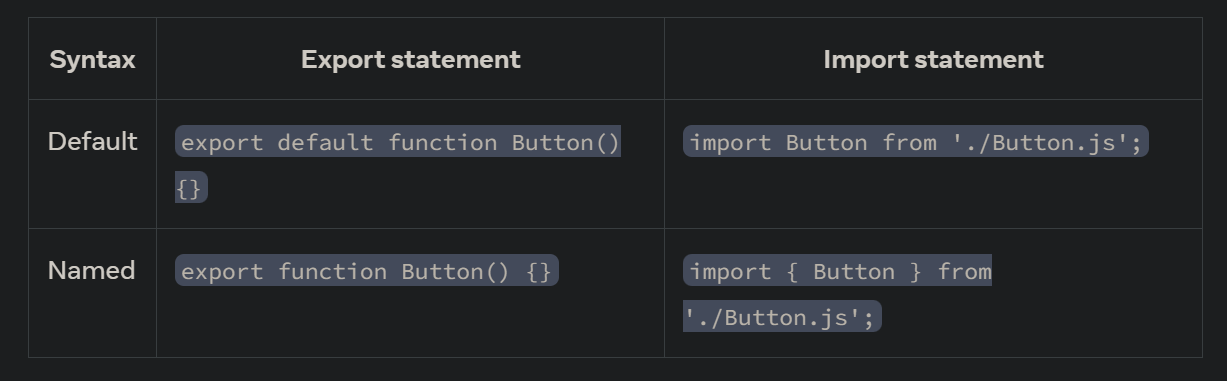
**'./Gallery.js':** This is the path to the file from which you are importing the **Gallery** module. The **./** at the beginning signifies that you are importing a file relative to the current file's location(matlab aile ko current file bata niska(.) ani tei folder ma gallery.js bhanne file bhitra jau bhaneko). In this case, it's importing the **Gallery.js** file from the same directory.

* If the file were in a different directory, you would adjust the path accordingly. For example, if it were in a **components** directory, the import might look like **import Gallery from './components/Gallery.js';**.
* The file extension **.js** is specified to indicate that it's a JavaScript file. However, in modern JavaScript and many build systems, including the file extension is often optional.
* Either './Gallery.js' or './Gallery' will work with React, though the former is closer to how [native ES Modules](https://developer.mozilla.org/docs/Web/JavaScript/Guide/Modules) work.

**Default vs named exports**

There are two primary ways to export values with JavaScript: default exports and named exports. So far, our examples have only used default exports. But you can use one or both of them in the same file. A file can have no more than one default export, but it can have as many named exports as you like.

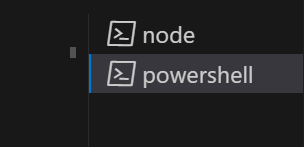
How you export your component dictates how you must import it. You will get an error if you try to import a default export the same way you would a named export! This chart can help you keep track:



When you write a *default* import, you can put any name you want after import. For example, you could write import Banana from './Button.js' instead and it would still provide you with the same default export. In contrast, with named imports, the name has to match on both sides. That’s why they are called *named* imports!

**People often use default exports if the file exports only one component, and use named exports if it exports multiple components and values.** Regardless of which coding style you prefer, always give meaningful names to your component functions and the files that contain them. Components without names, like export default () => {}, are discouraged because they make debugging harder.

NOTE:



Remember season dai said the importance of having two terminals,, it can be any but yes,,, because one terminal is for live server(local host), project gardai, site reload garna(ctrl + R to reload),,, ani arko terminal chai bich ma kei package install/uninstall haru garnu pare plus aru kei garna…..so basically you don’t have to end live server to install packages and stuff.

Browser ma dekhincha ni, that is render, so basically return statement bbhitra bhako sab kura browser ma dekhincha, so that all is render.

<Profile /> component lai esari arko file ma siddhai render garaidiye(return statement bhitra)……

import Profile from './profile.js';

esto lekhirakhnai parena, vs code le afai lekhdncha

## Recap

On this page you learned:

* What a root component file is
* How to import and export a component
* When and how to use default and named imports and exports
* How to export multiple components from the same file

**Writing Markup with JSX**

JSX is a syntax extension for JavaScript that lets you write HTML-like markup inside a JavaScript file. Although there are other ways to write components, most React developers prefer the conciseness of JSX, and most codebases use it.

**You will learn**

Why React mixes markup(html content that you wite within return tag) with rendering logic(component rendering)

How JSX is different from HTML

How to display information with JSX

## JSX: Putting markup into JavaScript

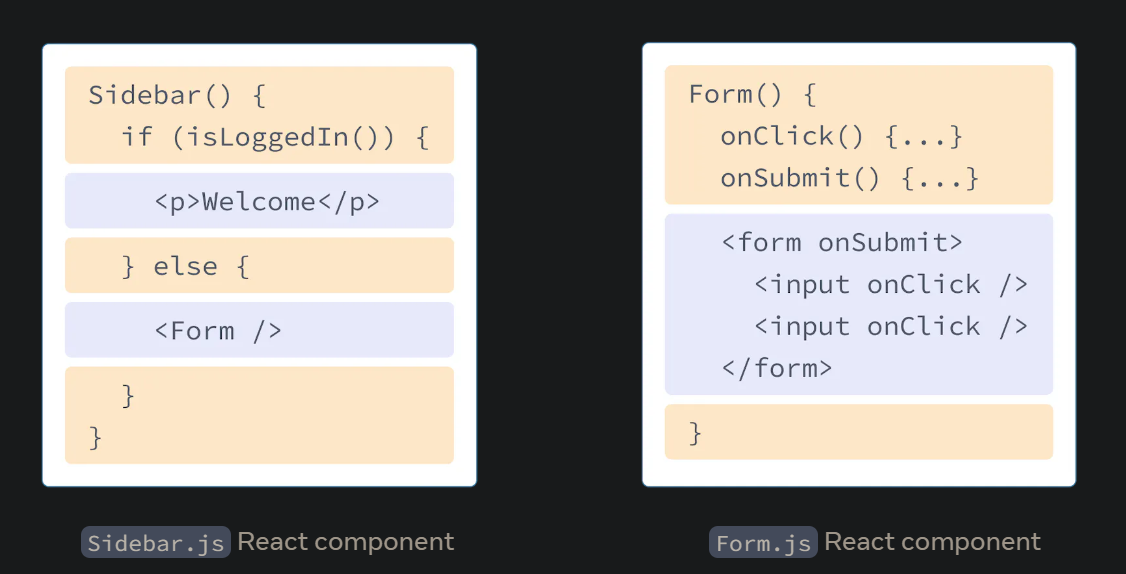
The Web has been built on HTML, CSS, and JavaScript. For many years, web developers kept content in HTML, design in CSS, and logic in JavaScript—often in separate files! Content was marked up inside HTML while the page’s logic lived separately in JavaScript:

But as the Web became more interactive, logic increasingly determined content. JavaScript was in charge of the HTML!

(Here are some key points to understand about how JavaScript interacts with HTML:

1. **DOM Manipulation:** JavaScript can manipulate the Document Object Model (DOM), which is a representation of the structure of an HTML document. The DOM is created by the browser when a web page is loaded. JavaScript allows you to dynamically modify, add, or remove HTML elements, attributes, and content on the page.
2. **Event Handling:** JavaScript can respond to user actions (such as clicks, keypresses, etc.) by attaching event listeners to HTML elements. When a specified event occurs, the associated JavaScript code is executed. This enables interactive features on a webpage.
3. **Dynamic Content:** JavaScript can be used to dynamically generate or modify HTML content based on user input, data from an external source, or other conditions. This dynamic content can enhance user experience by providing real-time updates and interactive features.

)This is why **in React, rendering logic and markup live together in the same place—components.**



Eg: Keeping a button’s rendering logic and markup together ensures that they stay in sync with each other on every edit. Conversely, details that are unrelated, such as the button’s markup and a sidebar’s markup, are isolated from each other, making it safer to change either of them on their own.

Each React component is a JavaScript function that may contain some markup that React renders into the browser. React components use a syntax extension called JSX to represent that markup. JSX looks a lot like HTML, but it is a bit stricter and can display dynamic information. The best way to understand this is to convert some HTML markup to JSX markup.

JSX and React are two separate things. They’re often used together, but you can use them independently of each other. JSX is a syntax extension, while React is a JavaScript library.

**The Rules of JSX**

**1. Return a single root element**

To return multiple elements from a component, wrap them with a single parent tag.

For example, you can use a <div>:

If you don’t want to add an extra <div> to your markup, you can write <> and </> instead:

This empty tag is called a Fragment. Fragments let you group things without leaving any trace in the browser HTML tree.(because <div> ya <section> use garyo bhane they are visible in your html dom tree ni so tyo browser tree ma nadekhauna <> - fragment use garincha).

(**Why do multiple JSX tags need to be wrapped?**

Hide Details

JSX looks like HTML, but under the hood it is transformed into plain JavaScript objects. You can’t return two objects from a function without wrapping them into an array. This explains why you also can’t return two JSX tags without wrapping them into another tag or a Fragment.))

**2. Close all the tags**

JSX requires tags to be explicitly closed: self-closing tags like <img> must become <img />, and wrapping tags like <li>oranges must be written as <li>oranges</li>.

So single tag haru lai ni tesaima close garnuparyo jsx ma. Eg. <img src=”pic1.jpg” />

<img src="https://i.imgur.com/yXOvdOSs.jpg" />

**3. camelCase all most of the things!**

JSX turns into JavaScript and attributes written in JSX become keys of JavaScript objects. In your own components, you will often want to read those attributes into variables. But JavaScript has limitations on variable names. For example, their names can’t contain dashes or be reserved words like class.

This is why, in React, many HTML and SVG attributes are written in camelCase. For example, instead of stroke-width you use strokeWidth. Since class is a reserved word, in React you write className instead, named after the [corresponding DOM property](https://developer.mozilla.org/en-US/docs/Web/API/Element/className):

<img

src="https://i.imgur.com/yXOvdOSs.jpg"

alt="Hedy Lamarr"

className="photo"

/>

You can [find all these attributes in the list of DOM component props.](https://react.dev/reference/react-dom/components/common) If you get one wrong, don’t worry—React will print a message with a possible correction to the [browser console.](https://developer.mozilla.org/docs/Tools/Browser_Console)

**Pitfall**

For historical reasons, [aria-\*](https://developer.mozilla.org/docs/Web/Accessibility/ARIA) and [data-\*](https://developer.mozilla.org/docs/Learn/HTML/Howto/Use_data_attributes) attributes are written as in HTML with dashes.

**NOTE:**

**<img src="https://i.imgur.com/yXOvdOSs.jpg" alt="Hedy Lamarr" className="photo" />**

**In this example, src, alt, and className are JSX attributes written in camelCase.**

1. **Reserved Words:**
   * **Certain words are reserved in JavaScript, and using them as variable or property names can lead to issues. For example, class is a reserved word in JavaScript. To avoid conflicts, React uses className instead of class when dealing with the class attribute in HTML elements. Id chai use garna milcha feri…class is a reserve word of html so mildaina.**
2. ***aria- and data-* Attributes:\*\***
   * **Attributes like aria-\* and data-\* are special attributes used for accessibility and custom data, respectively. Even in JSX, these attributes are written with dashes, following the traditional HTML convention.**

**Jsx Copy code**

**<div aria-labelledby="title" data-custom-attribute="value"> {/\* content \*/} </div>**

**In this example, aria-labelledby and data-custom-attribute are attributes using dashes.**

**EG of jsx code:**

export default function Bio() {

return (

<div>

<div id="d1" className="intro">

<h1>Welcome to my website!</h1>

</div>

<p className="summary">

You can find my thoughts here.

<br /><br />

<b>And <i>pictures</i></b> of scientists!

</p>

</div>

);

**}**

### Pro-tip: Use a JSX Converter

Converting all these attributes in existing markup can be tedious! We recommend using a [converter](https://transform.tools/html-to-jsx) to translate your existing HTML and SVG to JSX. Converters are very useful in practice, but it’s still worth understanding what is going on so that you can comfortably write JSX on your own.

**Recap**

**Now you know why JSX exists and how to use it in components:**

* **React components group rendering logic together with markup because they are related.**
* **JSX is similar to HTML, with a few differences. You can use a**[**converter**](https://transform.tools/html-to-jsx)**if you need to. HTML to jsx converter majale use garda huncha(cause dherai convert garnuparda tedious huncha) but concept bujeko hunuparcha.**
* **Error messages will often point you in the right direction to fixing your markup.**

**JavaScript in JSX with Curly Braces**

JSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to open a window to JavaScript.

**You will learn**

How to pass strings with quotes

How to reference a JavaScript variable inside JSX with curly braces

How to call a JavaScript function inside JSX with curly braces

How to use a JavaScript object inside JSX with curly braces

When you want to pass a string attribute to JSX, you put it in single or double quotes:

Eg: src="https://i.imgur.com/7vQD0fPs.jpg"

alt="Gregorio Y. Zara"

“<https://i.imgur.com/7vQD0fPs.jpg>” and "Gregorio Y. Zara" are string attributes passed to jsx, attribute names are src and alt.

But what if you want to dynamically specify the src or alt text? You could **use a value from JavaScript by replacing " and " with { and }**:

Eg:

export default function Avatar() {

const avatar = 'https://i.imgur.com/7vQD0fPs.jpg';

const description = 'Gregorio Y. Zara';

return (

<img

className="avatar"

src={avatar}

alt={description}

/>

);

}

Notice the difference between className="avatar", which specifies an "avatar" CSS class name that makes the image round, and src={avatar} that reads the value of the JavaScript variable called avatar. That’s because curly braces let you work with JavaScript right there in your markup!

**Using curly braces: A window into the JavaScript world**

JSX is a special way of writing JavaScript. That means it’s possible to use JavaScript inside it—with curly braces { }. The example below first declares a name for the scientist, name, then embeds it with curly braces inside the <h1>:

export default function TodoList() {

const name = 'Gregorio Y. Zara'; //yaha js ho hai yo, tara return bhitra ta sabai html content use hune ho, so tyata javascript use garnu pare ma use { }.

return (

<h1>{**name}**'s To Do List</h1>

**name** is a js variable so is written inside { } , when used inside the return statement.

);

}

Any JavaScript expression will work between curly braces, including function calls like **formatDate().**

Eg:

const today = new Date();

//This line creates a new Date object named today. The Date object represents the current date and time.

function formatDate(date) {

return new Intl.DateTimeFormat(

'en-US',

{ weekday: 'long' }

).format(date);

}

* //The **formatDate** function takes a **date** parameter and uses **Intl.DateTimeFormat** to format the date in a specific way.
* In this case, it formats the date to display the full name of the weekday in English (e.g., "Monday").

export default function TodoList() {

return (

<h1>To Do List for {formatDate(today)}</h1>

);

}

1.

* + The **TodoList** component is defined and exported as the default export of the module.
  + Inside the component, an **<h1>** element is returned. The content of this element includes the text "To Do List for" followed by the result of calling the **formatDate** function with the **today** date.
  + This means it dynamically generates a heading displaying the current day of the week in English.

1. **Using JSX to Insert Dynamic Content:**
   * JSX allows embedding JavaScript expressions inside curly braces **{}**. In this case, the result of **formatDate(today)** is inserted into the JSX expression, making the day of the week dynamic.

In summary, the **TodoList** component renders an **<h1>** element that displays a dynamic message, "To Do List for [current day of the week]," utilizing the **formatDate** function to format the date. The current date is obtained using the **Date** object created at the beginning of the code. This way, the displayed day of the week is always based on the current date when the component is rendered.

### Where to use curly braces

You can only use curly braces in two ways inside JSX:

1. **As text** directly inside a JSX tag: <h1>{name}'s To Do List</h1> works, but <{tag}>Gregorio Y. Zara's To Do List</{tag}> will not.
2. **As attributes** immediately following the = sign: src={avatar} will read the avatar variable, but src="{avatar}" will pass the string "{avatar}".

Using “double curlies”: CSS and other objects in JSX

Yo title bata continue studying next.

**Docs ko tippanis which I gottta include somewhere here:**

reach home, curly brces wala kura that dai said for img, static, named exports..........mero open react app ma img ko sourcing haruma bujheko kura haru pani include

sources – react documentation, afnai react app where I’m doing practically, chatgpt on the side to clear each and every doubt, mero wordfile note file on the side to jot anything and everything imp…..brainframe wordfile also open to keep reminiscing. Mobile rems eklot feri hera, mindset recall ramrari now that I’ve experienced the first day….luga ali patalo fukalna milne lagau please.

Image Source: Make sure the image source is correct and accessible. In your case, the source is a local file path (C:\Users\Lenovo\Desktop\react\src\pic1.jpg). When your React app is deployed, it won't have access to the file system of the user. You should import the image into your project and use a relative path or use a public URL.

For example, if you have the image in your project's public folder, you can reference it like this:

jsx

Copy code

<img src="/pic1.jpg" alt="Katherine Johnson" />

Camel case is a naming convention used in programming where the first letter of each word is capitalized except for the first word, which starts with a lowercase letter. The resulting style looks like the humps on a camel's back, hence the name "camel case." Here are a few examples to illustrate the concept:

In JavaScript, object keys are the names assigned to the properties of an object. Each key is a unique string or symbol that acts as an identifier for a specific property. Here's an overview of using keys in JavaScript objects:

### 1. String Keys:

Most commonly, object keys are strings. You can access and define properties using the dot notation or square bracket notation:

javascript

Copy code

// Using dot notation

const person = {

 name: "John",

 age: 30,

};

console.log(person.name); // Output: John

// Using square bracket notation

console.log(person["age"]); // Output: 30

### 2. Computed Property Names:

You can use expressions to compute property names in square bracket notation when creating or accessing object properties:

javascript

Copy code

const propertyName = "age";

const person = {

 name: "John",

 [propertyName]: 30,

};

console.log(person.age); // Output: 30

### 3. Symbol Keys:

Symbols can also be used as keys. Symbols are unique and can help avoid naming conflicts:

javascript

Copy code

const mySymbol = Symbol("mySymbol");

const obj = {

 [mySymbol]: "Hello!",

};

console.log(obj[mySymbol]); // Output: Hello!

### 4. Object Keys Method:

The Object.keys() method returns an array of a given object's own enumerable property names:

javascript

Copy code

const person = {

 name: "John",

 age: 30,

};

const keys = Object.keys(person);

console.log(keys); // Output: ['name', 'age']

### 5. Property Shorthand:

When creating an object, if the variable name is the same as the property name, you can use property shorthand:

javascript

Copy code

const name = "John";

const age = 30;

const person = {

 name,

 age,

};

console.log(person); // Output: { name: 'John', age: 30 }

These are some key concepts related to keys in JavaScript objects. Remember that keys in objects must be unique, and if you try to add a property with an existing key, it will overwrite the existing value. Additionally, keys are case-sensitive.

**Using “double curlies”: CSS and other objects in JSX**

In addition to strings, numbers, and other JavaScript expressions, you can even pass objects in JSX. Objects are also denoted with curly braces, like { name: "Hedy Lamarr", inventions: 5 }. Therefore, to pass a JS object in JSX, you must wrap the object in another pair of curly braces:

person={{ name: "Hedy Lamarr", inventions: 5 }}.

This is mostly seen while using inline CSS styles in JSX. React does not require you to use inline styles (CSS classes work great for most cases). But when you need an inline style, you pass an object to the style attribute:

export default function TodoList() {

return (

**<ul style={{**

**backgroundColor: 'black',**

**color: 'pink'**

**}}>**

<li>Improve the videophone</li>

<li>Prepare aeronautics lectures</li>

<li>Work on the alcohol-fuelled engine</li>

</ul>

);

}

**NOTE: Inline style properties are written in camelCase. For example, HTML <ul style="background-color: black"> would be written as <ul style={{ backgroundColor: 'black' }}> in your component.**

**The next time you see {{ and }} in JSX, know that it’s nothing more than an object inside the JSX curlies!**

**EXAMPLE of using CSS class with react**

import React from 'react';

import './MyComponent.css'; // Import the CSS file

const MyComponent = () => {

return (

<div className="my-component">

<h1>Hello, React!</h1>

<p>This is a simple React component.</p>

</div>

);

};

In this example, the **MyComponent** component has a **div** element with the **className** attribute set to **"my-component" , so yo div ko class “my-component” bhayo, ani esko css** MyComponent.css' yo file bata aairako cha jun lai top ma import pani gariyeko cha. This corresponds to a CSS class defined in a separate CSS file, which is imported at the top of the React component file.

/\* MyComponent.css \*/

.my-component {

background-color: #f0f0f0;

padding: 20px;

border-radius: 5px;

box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

}

.my-component h1 {

color: #333;

}

.my-component p {

color: #666;

}

In this CSS file, the **.my-component** class sets styles for the overall container, while **.my-component h1** and **.my-component p** target the **h1** and **p** elements inside the container, respectively.

React uses camelCase for its attributes to maintain consistency with JavaScript conventions. So **className** chai camel case ho, **my-component** haina.

**WHY WE MAKE OBJECTS IN JS/REACT? (copied from chatgpt)**

In JavaScript and React, creating objects is fundamental to organizing and managing data in your applications. Objects are a way to structure and group related data and functionality together. Here are some main reasons for creating objects in JavaScript and React:

1. **Data Organization:**
   * Objects allow you to organize related pieces of data into a single structure. For instance, you might have properties like **name**, **age**, **email** bundled together in an object representing a user.
2. **Encapsulation:**
   * Objects can encapsulate related behavior (methods/functions) along with data. This is known as encapsulation, and it helps keep related code and data together, making the code more modular and maintainable.
3. **Abstraction:**
   * Objects can abstract away complex implementation details. You can expose a simple interface (properties and methods) to the outside world while keeping the internal workings hidden.
4. **Code Reusability:**
   * Objects support code reusability. Once you've defined an object, you can create instances of it wherever needed, reducing redundancy in your code.
5. **Passing Complex Data:**
   * Objects are useful for passing complex data structures between functions and components. This is especially relevant in React, where components often receive and manage complex state objects.

In the context of React:

1. **Component State:**
   * React components often use objects to manage their state. State is an object that represents the current state of a component, and when the state changes, React re-renders the component.
2. **Props:**
   * Props, short for properties, are passed to React components as objects. They allow you to pass data from a parent component to a child component.
3. **API Responses:**
   * When dealing with data from APIs, you might often work with objects to represent the structure of the received data.

Here's a simple example of creating an object in JavaScript:

// Creating a user object

const user = {

name: 'John Doe',

age: 25,

email: 'john.doe@example.com',

sayHello: function() {

console.log(`Hello, my name is ${this.name}.`);

}

};

// Accessing properties and calling a method

console.log(user.name); // John Doe

user.sayHello(); // Hello, my name is John Doe.

In React, you'll frequently use objects to manage component state, props, and other data structures to represent the application's state.

Top of Form

Top of Form

**You can move several expressions into one object, and reference them in your JSX inside curly braces:**

**const person = {**

**name: 'Gregorio Y. Zara',**

**theme: {**

**backgroundColor: 'black',**

**color: 'pink'**

**}**

**};**

**export default function TodoList() {**

**return (**

**<div style={person.theme}>**

**<h1>{person.name}'s Todos</h1>**

**<img**

**className="avatar"**

**src="https://i.imgur.com/7vQD0fPs.jpg"**

**alt="Gregorio Y. Zara"**

**/>**

**<ul>**

**<li>Improve the videophone</li>**

**<li>Prepare aeronautics lectures</li>**

**<li>Work on the alcohol-fuelled engine</li>**

**</ul>**

**</div>**

**);**

**}**

The component **ToDoList** can use these values from **person** like so.

**CHALLENGE QUES THYO YO:**

**BE CAREFUL AT THE HIGHLIGHTED PART, HOW YOU SHOULD USE THEM**

const baseUrl = 'https://i.imgur.com/';

const person = {

name: 'Gregorio Y. Zara',

imageId: '7vQD0fP',

imageSize: 's',

theme: {

backgroundColor: 'black',

color: 'pink'

}

};

export default function TodoList() {

return (

<div style={person.theme}>

<h1>{person.name}'s Todos</h1>

<img

className="avatar"

**src= {baseUrl + person.imageId + person.imageSize + ".jpg"}**

**“.jpg” chai plain text ho, so esto concatenation ma careful hunuparyo. Yo chai js part ho, js variable src ma assign garirako ho hamile**

**Or, jsx ma return bhitra garnupare esari lekhnuparthyo <h1>{baseUrl}{person.imageId}{person.imageSize}.jpg</h1>**

alt={person.name}

/>

<ul>

<li>Improve the videophone</li>

<li>Prepare aeronautics lectures</li>

<li>Work on the alcohol-fuelled engine</li>

</ul>

</div>

);

}

**PASSING PROPS TO A COMPONENT**

React components use *props* to communicate with each other. Every parent component can pass some information to its child components by giving them props. you can pass any JavaScript value through them, including variable ko values, objects, arrays, and functions. A parent component is a component that contains and encapsulates other components. The parent component can pass data to its child components through props, manage the state that the child components depend on, and often controls the overall logic of the application.

Basically bhannuparda, Props are the information that you pass to a JSX tag. For example, className, src, alt, width, and height are some of the props you can pass to an <img>.

<**img**

**className="avatar"**

**src="https://i.imgur.com/1bX5QH6.jpg"**

**alt="Lin Lanying"**

**width={100}**

**height={100}**

**/>**

**The props you can pass to an <img> tag are predefined (ReactDOM conforms to** [**the HTML standard**](https://www.w3.org/TR/html52/semantics-embedded-content.html#the-img-element)**).**

**But you can pass any props to *your own* components also…let’s see how**

In this code, the Profile component isn’t passing any props to its child component, Avatar:

**export default function Profile() {**

**return (**

**<Avatar />**

**);**

**}**

You can give Avatar some props in two steps.

**export default function Profile() {**

**return (**

**<Avatar**

**person={{ name: 'Lin Lanying', imageId: '1bX5QH6' }}**

**size={100}**

**/>**

**);**

**}**

**function Avatar({ person, size }) {**

**// person and size are available here**

**}**

Don’t miss the pair of { and } curlies inside of ( and ) when declaring props:

**function Avatar({ person, size }) {**

**// ...**

**}**

This syntax is called [“destructuring”](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Unpacking_fields_from_objects_passed_as_a_function_parameter) and is equivalent to reading properties from a function parameter:

**function Avatar(props) {**

**let person = props.person;**

**let size = props.size;**

**// ...**

**}**

If you want to give a prop a default value to fall back on when no value is specified, you can do it with the destructuring by putting = and the default value right after the parameter:

function Avatar({ person, size = 100 }) {

// ...

}

Now, if <Avatar person={...} /> is rendered with no size prop, the size will be set to 100.

The default value is only used if the size prop is missing or if you pass size={undefined}. But if you pass size={null} or size={0}, the default value will not be used.

PROPS( what I’ve understood):

-props stand for properties hai

-<img src=… alt =…> , yaha src and alt also are props. They are predefined props, right.

-Aba we can make our own props as well. So props chai basically parent component bata child component lai pass garincha when the child component is being rendered inside the parent component.

**Eg.:**

**export default function App(){ //App is parent component**

**…..**

**….**

**<Avatar //Avatar is child component**

**person={{name=”Raj”, //person is an object, we’re passing it as prop**

**age=40,**

**vehicle=”car”}}**

**color={“red”} //color ni prop pass gareko, string lai double quotes**

**companies={4} /> //companies ni prop, this is a number**

**}**

**-**Ani aba yo pass gariyeko props, child component “Avatar” le n accept garnuparcha esari:

export default function Avatar ({person, color, companies}) {

…..

<aba eta yo Avatar bhitra person, color ra companies normally as a variable nai use garna milcha…..ani yei ko kura display garauna chai {} bhitra rakhnuparcha,,,I’ll you you an example and you’ll understand ok

export default function Button({color, text}){

    return(

    <>

    <button style={{ backgroundColor: color }}>{text}</button>

    </>)

}

Eta hera, style bhitra color assign garda etikai use bhako cha, but display garnuparda {text}, esari lekheka cham.

NOW,

export default function Button({color, text}){

THIS IS DESTRUCTURING PRACTISE AND REGARDED GOOD

export default function Button({props}){

Natra, esto ni garna milthyo, tara feri, jata ni aba props.color, props.text bhanera lekhnuparthyo,, so jata ni props props bhayera debugging garna garo huncha, so destructuring is a good practice.

Esari props ko rup ma accept garda, react le props bhanne euta object banaidincha ani parent le pathako savbai props tyo object ko props banaidincha(remember we said predefined props? Properties banaidincha object ko basically, ani props object bata access garnuparcha: props.text, props.color garera..

But mind you, destructuring is always the best approach.

SEASON DAI {} HE SAID fr constants(static), js variable or anything js inside return right

ANI AFNO QUERY REMEMBER OF OBJECT MA =, STYLE BHITRA:, HTML MA =, OBJECT KO SYNTAX IN GENERAL, INSIDE JSX, BACKGROUNDCOLOR KATA KASARI LEKHNE, sab thau ma backgroundColor nai lekhne ho hai now in attributes “ ” bhitra matra ni j ni lekhna milcha

**CONDITIONAL RENDERING: (STH TO REMEMBER)-STRING INTERPOLATION while using logical and (&&) KO KURA**

function Item({ name, importance }) {

return (

<li className="item">

{name}{importance>0 && ` (Importance: ${importance} )`}

</li>

);

}

This code uses the ternary operator to conditionally render the importance part only if **importance** is greater than 0. The template literals (**${}**) are used for string interpolation to include the dynamic value of **importance**. When using template literals in JavaScript, you should use backticks (`) instead of double quotes.

export default function PackingList() {

return (

<section>

<h1>Sally Ride's Packing List</h1>

<ul>

<Item

importance={9}

name="Space suit"

/>

<Item

importance={0}

name="Helmet with a golden leaf"

/>

<Item

importance={6}

name="Photo of Tam"

/>

</ul>

</section>

);

}

**NOTE:**

**WHAT is children?**

In React, **children** is a special prop that is used to pass components or elements as children to another component. It allows you to nest content inside a component in a flexible and dynamic way.

When you create a component and use it in JSX, anything between the opening and closing tags of that component becomes the **children** of that component. For example:

**// ParentComponent.jsx**

function ParentComponent({ children }) {

return (

<div>

{children}

</div>

);

}

**// App.jsx**

function App() {

return (

<ParentComponent>

<h1>Hello, World!</h1>

<p>This is a child element.</p>

</ParentComponent>

);

}

In this case, the content "Click me!" passed to the **Button** component becomes the **children** prop, and it is rendered inside the **<button>** element.

Using **children** allows you to create more flexible and reusable components that can accept and render different content based on the needs of the parent component.

**Passing event handlers as props**

Often you’ll want the parent component to specify a child’s event handler. Consider buttons: depending on where you’re using a Button component, you might want to execute a different function—perhaps one plays a movie and another uploads an image.

To do this, pass a prop the component receives from its parent as the event handler like so:

**function Button({ onClick, children }) {**

**return (**

**<button onClick={onClick}>**

**{children}**

**</button>**

**);**

**}**

**function PlayButton({ movieName }) {**

**function handlePlayClick() {**

**alert(`Playing ${movieName}!`);**

**}**

**return (**

**<Button onClick={handlePlayClick}>**

**Play "{movieName}"**

**</Button>**

**);**

**}**

**function UploadButton() {**

**return (**

**<Button onClick={() => alert('Uploading!')}>**

**Upload Image**

**</Button>**

**);**

**}**

**export default function Toolbar() {**

**return (**

**<div>**

**<PlayButton movieName="Kiki's Delivery Service" />**

**<UploadButton />**

**</div>**

**);**

**}**

**NOTE TO TAKE:**

**Event handlers will also catch events from any children your component might have. We say that an event “bubbles” or “propagates” up the tree: it starts with where the event happened, and then goes up the tree.**

Here, the Toolbar component renders a PlayButton and an UploadButton:

* PlayButton passes handlePlayClick as the onClick prop to the Button inside.
* UploadButton passes () => alert('Uploading!') as the onClick prop to the Button inside.

Finally, your Button component accepts a prop called onClick. It passes that prop directly to the built-in browser <button> with onClick={onClick}. This tells React to call the passed function on click.

If you use a [design system](https://uxdesign.cc/everything-you-need-to-know-about-design-systems-54b109851969), it’s common for components like buttons to contain styling but not specify behavior. Instead, components like PlayButton and UploadButton will pass event handlers down.

**Event propagation**

Event handlers will also catch events from any children your component might have. We say that an event “bubbles” or “propagates” up the tree: it starts with where the event happened, and then goes up the tree.

This <div> contains two buttons. Both the <div> *and* each button have their own onClick handlers. Which handlers do you think will fire when you click a button?

export default function Toolbar() {

return (

<div className="Toolbar" onClick={() => {

alert('You clicked on the toolbar!');

}}>

<button onClick={() => alert('Playing!')}>

Play Movie

</button>

<button onClick={() => alert('Uploading!')}>

Upload Image

</button>

</div>

);

}

If you click on either button, its onClick will run first, followed by the parent <div>’s onClick. So two messages will appear. If you click the toolbar itself, only the parent <div>’s onClick will run.

All events propagate in React except onScroll, which only works on the JSX tag you attach it to.

You could add more code to this handler before calling the parent onClick event handler, too. This pattern provides an *alternative* to propagation. It lets the child component handle the event, while also letting the parent component specify some additional behavior. Unlike propagation, it’s not automatic. But the benefit of this pattern is that you can clearly follow the whole chain of code that executes as a result of some event.

If you rely on propagation and it’s difficult to trace which handlers execute and why, try this approach instead.

**Preventing default behavior**

Some browser events have default behavior associated with them. For example, a <form> submit event, which happens when a button inside of it is clicked, will reload the whole page by default:

You can call e.preventDefault() on the event object to stop this from happening:

export default function Signup() {

return (

<form onSubmit={e => {

e.preventDefault();

alert('Submitting!');

}}>

<input />

<button>Send</button>

</form>

);

}

Don’t confuse e.stopPropagation() and e.preventDefault(). They are both useful, but are unrelated:

* [e.stopPropagation()](https://developer.mozilla.org/docs/Web/API/Event/stopPropagation) stops the event handlers attached to the tags above from firing.
* [e.preventDefault()](https://developer.mozilla.org/docs/Web/API/Event/preventDefault)prevents the default browser behavior for the few events that have it.
* **Preventing Default Behavior:**
  + The **e.preventDefault()** method is called within the **onSubmit** event handler.
  + This method prevents the default behavior of the form, which is to submit the data to the server and reload the page.( In a typical HTML form without any JavaScript intervention, when a user clicks the "Submit" button or presses Enter inside an input field, the form is submitted to the server.)
  + By preventing the default behavior, the form data is not sent to the server, and the page does not refresh.
* **Alert Message:**
  + After preventing the default behavior, an **alert** is triggered with the message 'Submitting!'.
  + This is a placeholder action, and typically, in a real-world scenario, you would perform further actions like client-side validation, sending data to an API, or navigating to another page using client-side routing.
* **Client-Side Handling:**
  + By preventing the default form submission behavior, developers gain control over how the form data is handled on the client side.
  + JavaScript code can be executed to validate inputs, make asynchronous requests, or perform other client-side actions before deciding whether to proceed with submitting the form.

In summary, the **e.preventDefault()** prevents the default behavior of form submission, allowing developers to handle the form data in a custom way using client-side JavaScript logic. This is often used in modern web applications to provide a smoother and more interactive user experience without causing full page reloads.

**Can event handlers have side effects?**

Absolutely! Event handlers are the best place for **side effects**.

Unlike rendering functions, event handlers don’t need to be [pure](https://react.dev/learn/keeping-components-pure), so it’s a great place to *change* something—for example, change an input’s value in response to typing, or change a list in response to a button press. However, in order to change some information, you first need some way to store it. **In React, this is done by using** [**state, a component’s memory.**](https://react.dev/learn/state-a-components-memory) **You will learn all about it on the next page.**

**NOTE: (Acc. to chatgpt)**

In programming**, a side effect** refers to any modification of state or behavior that is observable outside of a function or a block of code. **In simpler terms, a side effect is any action or change that occurs as a result of executing a piece of code, other than simply returning a value.**

Side effects can be broadly categorized into two types:

1. **Observable State Changes:**
   * Modifying variables, objects, or data structures outside the scope of a function(eg.: jun event handler function use gariracham, any variable outside this function ko value change gardine).
   * For example, changing the value of a global variable, updating an object in the state, or modifying elements in an array.
2. **External Interactions:**
   * Interacting with the external environment, **such as reading from or writing to a file, making network requests, or updating the DOM in a web application.**
   * For example, **fetching data from an API, saving data to a database, or manipulating the DOM to update the user interface.**

In the context of React event handlers, side effects are often necessary and expected. For example:

* **Modifying State:**
  + When handling user interactions, you often need to update the state of a component to reflect changes in the user interface. **Modifying state is a side effect.**
* **Asynchronous Operations:**
  + **Performing asynchronous operations, like fetching data from an API, is a common side effect in React components.** This is often done using functions like **fetch** or **axios** within event handlers.
* **DOM Manipulation:**
  + **Changing the structure or appearance of the DOM (Document Object Model) in response to user interactions is also a side effect. This includes tasks like showing/hiding elements, updating styles, etc.**

In functional programming, there is a preference for pure functions (functions without side effects), as they are easier to test and reason about. However, in real-world applications, side effects are often unavoidable and necessary for achieving certain functionality.

In the provided quote, when it mentions that event handlers are a great place for side effects, it means that event handlers are well-suited for performing actions or changes in response to user interactions, as this is a common and expected use case in web development.

**RECAP:**

* Events propagate upwards. Call e.stopPropagation() on the first argument to prevent that.
* Events may have unwanted default browser behavior. Call e.preventDefault() to prevent that.
* Explicitly calling an event handler prop from a child handler is a good alternative to propagation.

**NOTE:**

**Challenge 4 of 4:**

Remove unnecessary state

When the button is clicked, this example should ask for the user’s name and then display an alert greeting them. You tried to use state to keep the name, but for some reason it always shows “Hello, !“.

To fix this code, remove the unnecessary state variable. (We will discuss about [why this didn’t work](https://react.dev/learn/state-as-a-snapshot) later.)

Can you explain why this state variable was unnecessary?

**A state variable is only necessary to keep information between re-renders of a component. Within a single event handler, a regular variable will do fine. Don’t introduce state variables when a regular variable works well.**

NOTE:

IN VANILLA JS OR JAVASCRIPT PART OF JSX(IN REACT)KO SANDARVA MA

alert(`Hello, ${name}!`);

The **${name}** is a placeholder for the value of the **name** variable. It is part of a template literal, denoted by the backticks (**``**), which allows you to embed expressions (in this case, the value of **name**) within a string.

**Template literals are a feature introduced in ECMAScript 6 (ES6) that allow you to embed expressions within string literals using backticks ( ) as delimiters. The expressions inside the ${} notation are evaluated, and their results are inserted into the string.**

**In your example:**

* **${name} is the expression inside the template literal.**
* **The value of the name variable is inserted into the string where ${name} is located.**

**This provides a more concise and readable way to concatenate variables and expressions within strings compared to traditional string concatenation or using the + operator.**

**Here's a brief comparison:**

**Traditional String Concatenation:**

**alert("Hello, " + name + "!");**

**Template Literal:**

**alert(`Hello, ${name}!`);**

Using template literals is generally preferred for its readability and ease of use, especially when dealing with complex string concatenation or multiline strings.

**NOTE TO UNDERSTAND:**

Yes, you are correct. In React, the state updates triggered by functions like **setNumber** are asynchronous. When you call **setNumber(number + 1)**, React doesn't immediately update the state and re-render the component. Instead, it schedules a re-render, and the state update occurs before the next render.

In the example you provided:

<button onClick={() => {

setNumber(number + 1);

setNumber(number + 1);

setNumber(number + 1);

}}>+3</button>

All three calls to **setNumber** are executed in the same event loop, and the **number** value is still the same (unchanged) until the next render. So, when you click the button, React schedules a re-render **after the entire event handler has finished**. All three calls to **setNumber** are based on the same **number**, which is the old value.

**React waits until all code in the event handlers has run before processing your state updates.** This lets you update multiple state variables—even from multiple components—without triggering too many [re-renders.](https://react.dev/learn/render-and-commit#re-renders-when-state-updates) But this also means that the UI won’t be updated until *after* your event handler, and any code in it, completes. This behavior, also known as batching, makes your React app run much faster.

## Updating the same state multiple times before the next render

It is an uncommon use case, but if you would like to update the same state variable multiple times before the next render, instead of passing the next state value like setNumber(number + 1), you can pass a function that calculates the next state based on the previous one in the queue, like setNumber(n => n + 1). It is a way to tell React to “do something with the state value” instead of just replacing it.

import { useState } from 'react';

export default function Counter() {

const [number, setNumber] = useState(0);

return (

<>

<h1>{number}</h1>

<button onClick={() => {

setNumber(n => n + 1);

setNumber(n => n + 1);

setNumber(n => n + 1);

}}>+3</button>

</>

)

}

Here, n => n + 1 is called an **updater function.** (arrow function ho ni, function ma n is oming as a parameter, website refer once properly, you’ll understand) When you pass it to a state setter:

1. React queues this function to be processed after all the other code in the event handler has run.
2. During the next render, React goes through the queue and gives you the final updated state.

**ES BHITRA AJHAI BUJHNUPARNE KURA CHA PLEASE RECAP IT FROM THE REACT KAI WEBSITE UNDER TOPIC: QUEUEING A SERIES OF STATE UPDATES…**

**NOTE:**

**What if you put a timer on the alert, so it only fires *after* the component re-rendered? Would it say “0” or “5”? Have a guess!**

import { useState } from 'react';

export default function Counter() {

const [number, setNumber] = useState(0);

return (

<>

<h1>{number}</h1>

<button onClick={() => {

setNumber(number + 5);

setTimeout(() => {

alert(number);

}, 3000);

}}>+5</button>

</>

)

}

The state stored in React may have changed by the time the alert runs, but it was scheduled using a snapshot of the state at the time the user interacted with it!

**A state variable’s value never changes within a render,** even if its event handler’s code is asynchronous. Inside *that render’s* onClick, the value of number continues to be 0 even after setNumber(number + 5) was called. Its value was “fixed” when React “took the snapshot” of the UI by calling your component.

# Updating Objects in State

State can hold any kind of JavaScript value, including objects. But you shouldn’t change objects that you hold in the React state directly. Instead, when you want to update an object, you need to create a new one (or make a copy of an existing one), and then set the state to use that copy.

React encourages treating state values as immutable, even for objects. While it's technically possible to mutate an object directly in React state, doing so can lead to unintended consequences and issues with the component's rendering.

The recommended approach is to use the **useState** function's updater form, which provides a way to update state based on the previous state. This ensures that you're always working with the latest state and helps avoid potential race conditions.

By treating objects in state as if they were immutable, you help maintain a clear and predictable flow of data in your React components, reducing the risk of bugs and making it easier to reason about your application's state changes.

In JavaScript, primitive values such as numbers, strings, and booleans are indeed immutable. When you assign a new value to a variable, you're not modifying the existing value; instead, you're creating a new value. For example:

let x = 0; x = 5;

// You're not modifying the number 0; you're assigning a new value (5) to the variable x

Regarding objects, I want to clarify that the term "mutable" refers to the ability to change the internal state of an object, including adding, modifying, or deleting properties. In JavaScript, objects are mutable, meaning you can modify their properties directly. For instance:

let obj = { x: 0, y: 0 }; obj.x = 5; // Modifying the value of the property x

However, in the context of React state management, it's recommended to treat objects as if they were immutable for reasons related to how React handles state updates and re-renders.

To achieve immutability with objects in React, you often create a new object based on the previous state when updating,,,you’ll learn how in react.dev

In JavaScript, when we talk about immutability, we are referring to the concept that values themselves cannot be changed once they are created. Primitive types such as numbers, strings, and booleans are immutable in the sense that the values themselves cannot be modified. When you perform an operation that seems to change a number, you are actually creating a new value, and the original value remains unchanged.

For example:

let x = 0;

x = 5; // The value of x is now 5, but the original value 0 is not modified.

In contrast, objects in JavaScript are mutable, meaning their properties can be changed directly. When you modify a property of an object, you are altering the existing object in memory:

let obj = { x: 0, y: 0 };

obj.x = 5;

// Modifying the existing object by changing the value of the property x

Simply bhanna khoje kura chai yedi pail x=0 garethyam ani aile feri x=5 garyam bhane tyo hamile x=0 lai nai modify garera 5 banako haina, siddhai feri x bhanne variable banayera 5 rakhdeko ani tesle paila ko x=0 lai replace gardeko ho…

But in object

Suppose

Let object1={name:”Aashra”,

age: 19};

aba we can do, object1.age=20, yaha hamile feri arko object banayera replace ta gareko haina ni, just tei paila kai exist garirako object ko euta property ko value lai modify gareko. So yo concept bujhnuparyo……ANI EVEN THOUGH THIS IS TECHNICALLY POSSIBLE, OBJECTS IN REACT ARE MUTABLE,,,, TAIPANI THIS ISN’T A NICE APPROACH ani objects lai ni immutable jasari nai sochera, kei modify garnupare, arko object banayera assign garnuparcha, instead of simply modifying the properties of object directly…..ESTO taba sochne(that objects are immutable) jaba hamile state variable ma object store gariracham. Instead of mutating them, you should always replace them. when you want to update an object, you need to create a new one (or make a copy of an existing one), and then set the state to use that copy. In other words, you should treat **any JavaScript object that you put into state** as read-only**. ARU BELA OBJECT NORMALLY USE GARDA(NOT AS STATE),DOING MUTATION IS COMPLETELY OK!**

However, when working with React state, even though JavaScript allows direct mutation of object properties, it is recommended to treat objects as if they were immutable. This is because React relies on the concept of immutability to efficiently update the virtual DOM and trigger re-renders.

So, while you can directly modify an object's properties in JavaScript, the recommendation in React is to create new objects when updating state to ensure a more predictable and manageable state management flow. This is often achieved using techniques like the spread operator (**{ ...prevObject, newProperty: newValue }**) or libraries like Immer or Immutable.js.

Top of Form

NOW NOTE:

Top of Form

The problem is with this bit of code.

onPointerMove={e => {

position.x = e.clientX;

position.y = e.clientY;

}}

This code modifies the object assigned to position from the previous render. But without using the state setting function, React has no idea that object has changed. So React does not do anything in response. While mutating(modifying) state can work in some cases, we don’t recommend it. You should treat the state value you have access to in a render as read-only.

To actually [trigger a re-render](https://react.dev/learn/state-as-a-snapshot#setting-state-triggers-renders) in this case, **create a *new* object and pass it to the state setting function:**

onPointerMove={e => {

setPosition({

x: e.clientX,

y: e.clientY

});

}}

With setPosition, you’re telling React:

* Replace position with this new object
* And render this component again

**Local mutation is fine**

Code like this is a problem because it modifies an existing object in state:

position.x = e.clientX;

position.y = e.clientY;

But code like this is absolutely fine because you’re mutating a fresh object you have just created:

const nextPosition = {};

nextPosition.x = e.clientX;

nextPosition.y = e.clientY;

setPosition(nextPosition);

In fact, it is completely equivalent to writing this:

setPosition({

x: e.clientX,

y: e.clientY

});

Mutation is only a problem when you change existing objects that are already in state. Mutating an object you’ve just created is okay because no other code references it yet. Changing it isn’t going to accidentally impact something that depends on it. This is called a “local mutation”. You can even do local mutation while rendering. Very convenient and completely okay!

## **Copying objects with the spread syntax**

You can use the ... [object spread](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_object_literals) syntax so that you don’t need to copy every property separately.

Like this:

setPerson({

...person, // Copy the old fields

firstName: e.target.value // But override this one

});

NATRA esto garnuparthyo:

setPerson({

firstName: e.target.value, // New first name from the input

lastName: person.lastName,

email: person.email

});

**Updating a nested object**

Consider a nested object structure like this:

const [person, setPerson] = useState({

name: 'Niki de Saint Phalle',

artwork: {

title: 'Blue Nana',

city: 'Hamburg',

image: 'https://i.imgur.com/Sd1AgUOm.jpg',

}

});

In order to change city, you would first need to produce the new artwork object (pre-populated with data from the previous one), and then produce the new person object which points at the new artwork:

const nextArtwork = { ...person.artwork, city: 'New Delhi' };

const nextPerson = { ...person, artwork: nextArtwork };

setPerson(nextPerson);

**Objects are not really nested**

An object like this appears “nested” in code:

let obj = {

name: 'Niki de Saint Phalle',

artwork: {

title: 'Blue Nana',

city: 'Hamburg',

image: 'https://i.imgur.com/Sd1AgUOm.jpg',

}

};

However, “nesting” is an inaccurate way to think about how objects behave. When the code executes, there is no such thing as a “nested” object. You are really looking at two different objects:

let obj1 = {

title: 'Blue Nana',

city: 'Hamburg',

image: 'https://i.imgur.com/Sd1AgUOm.jpg',

};

let obj2 = {

name: 'Niki de Saint Phalle',

artwork: obj1

};

The obj1 object is not “inside” obj2. For example, obj3 could “point” at obj1 too:

let obj1 = {

title: 'Blue Nana',

city: 'Hamburg',

image: 'https://i.imgur.com/Sd1AgUOm.jpg',

};

let obj2 = {

name: 'Niki de Saint Phalle',

artwork: obj1

};

let obj3 = {

name: 'Copycat',

artwork: obj1

};

If you were to mutate obj3.artwork.city, it would affect both obj2.artwork.city and obj1.city. This is because obj3.artwork, obj2.artwork, and obj1 are the same object. This is difficult to see when you think of objects as “nested”. Instead, they are separate objects “pointing” at each other with properties.

**IMMER**

Notice how much more concise the event handlers have become. You can mix and match useState and useImmer in a single component as much as you like. Immer is a great way to keep the update handlers concise, especially if there’s nesting in your state, and copying objects leads to repetitive code.

#### Why is mutating state not recommended in React?

There are a few reasons:

* **Debugging:** If you use console.log and don’t mutate state, your past logs won’t get clobbered by the more recent state changes. So you can clearly see how state has changed between renders.
* **Optimizations:** Common React [optimization strategies](https://react.dev/reference/react/memo) rely on skipping work if previous props or state are the same as the next ones. If you never mutate state, it is very fast to check whether there were any changes. If prevObj === obj, you can be sure that nothing could have changed inside of it.
* **New Features:** The new React features we’re building rely on state being [treated like a snapshot.](https://react.dev/learn/state-as-a-snapshot) If you’re mutating past versions of state, that may prevent you from using the new features.
* **Requirement Changes:** Some application features, like implementing Undo/Redo, showing a history of changes, or letting the user reset a form to earlier values, are easier to do when nothing is mutated. This is because you can keep past copies of state in memory, and reuse them when appropriate. If you start with a mutative approach, features like this can be difficult to add later on.
* **Simpler Implementation:** Because React does not rely on mutation, it does not need to do anything special with your objects. It does not need to hijack their properties, always wrap them into Proxies, or do other work at initialization as many “reactive” solutions do. This is also why React lets you put any object into state—no matter how large—without additional performance or correctness pitfalls.

In practice, you can often “get away” with mutating state in React, **but we strongly advise you not to do that so that you can use new React features developed with this approach in mind. Future contributors and perhaps even your future self will thank you!**

NOTE to recall:

* When you store objects in state, mutating them will not trigger renders and will change the state in previous render “snapshots”.
* Instead of mutating an object, create a *new* version of it, and trigger a re-render by setting state to it.

NOTE: (highlighted , lai ekdam yaad garnu hai syntax ma)

import { useState } from 'react';

export default function Scoreboard() {

const [player, setPlayer] = useState({

firstName: 'Ranjani',

lastName: 'Shettar',

score: 10,

});

function handlePlusClick() {

setPlayer({...player, score:(player.score+1),});

}

function handleFirstNameChange(e) {

setPlayer({

...player,

firstName: e.target.value,

});

}

function handleLastNameChange(e) {

setPlayer({

...player,

lastName: e.target.value,

});

}

return (

<>

<label>

Score: <b>{player.score}</b>

{' '}

<button onClick={handlePlusClick}>

+1

</button>

</label>

<label>

First name:

<input

value={player.firstName}

onChange={handleFirstNameChange}

/>

</label>

<label>

Last name:

<input

value={player.lastName}

onChange={handleLastNameChange}

/>

</label>

</>

);

}

Top of Form