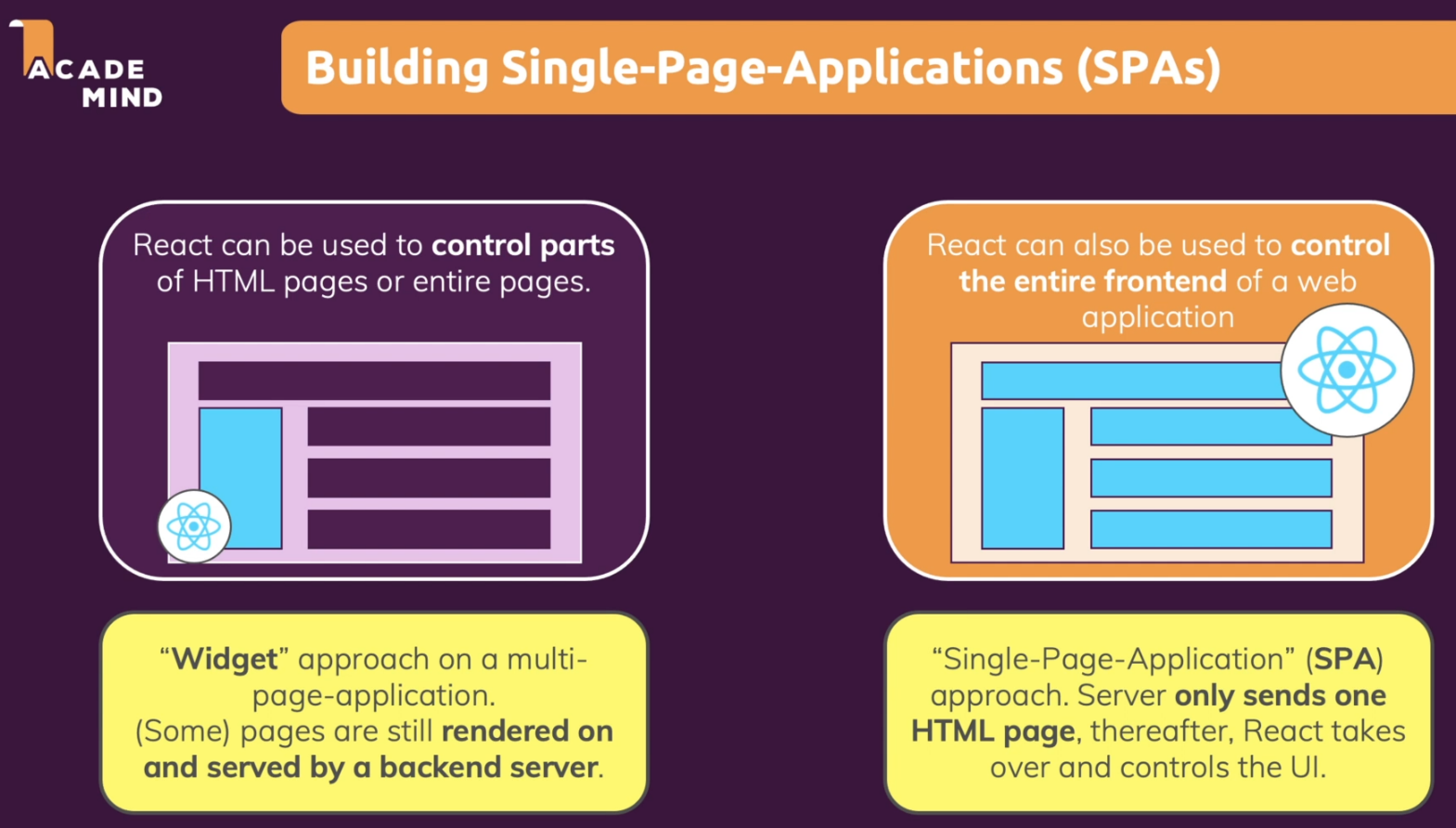
**React Notes**

React

* React is a client – side JavaScript library for building modern, reactive user interface for the Web using declarative, component – focused approach.
* JavaScript is a programming language that allows a developer to run logic in the browser.
* JavaScript runs in the browser – on the loaded page. We can manipulate the HTML Structure (DOM) of the page.
* When working with React, we often build so-called Single Page Applications.



Topics

1. React Basics & Working with Components
   1. Components
   2. Declarative & Imperative ways
   3. React project creation
   4. Intro to JSX (JavaScript XML)
   5. Custom Component
   6. Props concept
   7. Splitting Components into Multiple Components
   8. Composition concept
   9. Alternative Function syntax (Arrow function)
2. React State & Working with Events
   1. Listening to events & working with events Handlers
   2. Component function Execution
   3. Working with “State” & “useState” Hook
   4. Ways to update state
   5. Adding Form inputs, Listening to User Input & Handling form submission
   6. Working with Multiple states Vs Using One state
   7. Updating state that depends on the previous state
   8. Adding 2-way binding
   9. Child to Parent Component Communication (Bottom-up), Lifting the State Up
   10. Controlled Vs Uncontrolled Components, Stateless Vs Stateful Components
3. Rendering Lists & Conditional Content
   1. Rendering List of Data
   2. Using Stateful Lists
   3. Understanding “Keys”
   4. Outputting conditional content
   5. Adding conditional return statements
   6. Adding Dynamic styles

**1. React Basics & Working with Components**

**1.1. Components**

* React applications are built from isolated pieces of UI called Components.
* A component is a piece of the UI (User interface) that has its own logic & appearance.
* A component can be as small as a button, or as large as an entire page.
* React components are JavaScript functions that return markup (HTML or JSX)
* React components are regular JavaScript functions except:

1. Their names always begin with a capital letter.
2. They return JSX markup.

|  |
| --- |
| function MyButton() {    return (<button>I'm a button</button>);  }  export default function MyApp() {    return (      <div>        <h1>Welcome to my app</h1>        <MyButton />      </div>    );  } |

* Why Components?
* **Reusability** (Having reusable building blocks helps us avoid repetition)
* **Separation of Concerns** (Having a separation of concerns helps us with keeping our code base small & manageable instead of having one large file which holds all the HTML code & all the JavaScript logic.
* Recommended Approach
* For the entire user interface, we should create small separated units or components & each component should have one clear concern, one focus, one specific task that it should focus on.

**Steps for defining a component**

**Step 1:** Export the component

* The **export default** prefix is a standard JavaScript syntax.
* It lets us mark the main function in a file so that we can later import it from other files.

**Step 2:** Define the function

* React components are regular JavaScript functions, but their names must start with a Capital letter or they won’t work.
* We can define component using “function” keyword or using “arrow” function.

**Step 3:** Add markup

* The components return the JSX i.e., embedded XML inside JavaScript.
* We must wrap the JSX code in a pair of parentheses.

**Nesting & organizing components**

* Components are regular JavaScript functions, so we can keep multiple components in the same file.
* Only the main component in a file is declared with export default. Others must be declared as normal functions.
* We can export our function component from the source file to destination using either **default export** or **named exports.**

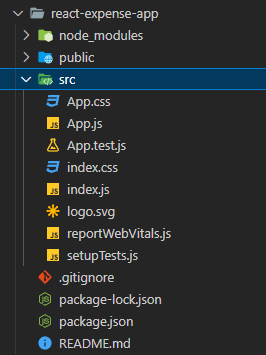
|  |
| --- |
| import Gallery from './Gallery.js'; // default export (valid for main component)  import { Profile } from './Gallery.js'; // Named export (valid for other than main component) |

* This is convenient when components are relatively small or tightly related to each other.
* We can create separate components to promote loose coupling & then import those components based on requirements.
  1. **Declarative & Imperative ways**
* React allows you to create re – usable & reactive components consisting of HTML & JavaScript (and CSS)
* React uses something called a declarative approach for building these components.
* In Imperative programming approach, we need to write the exact step by step instructions to manipulate the UI depending on what just happened i.e., we have to “command” each element telling the computer how to update the UI.
* In Declarative programming approach, we describe the UI for each visual state rather than micromanaging the UI (imperative).
* **React follows declarative approach i.e., define the desired target state(s) & let React figure out the actual JavaScript DOM instructions.**
* Declarative approach is basically mean that with React, we will not tell React that a certain HTML element should be created & inserted in a specific place on the UI as we would be doing that with Vanilla JavaScript.
* Instead with React, we will always define the desired end state (the target state) or conditional target state & it’s then React’s job to figure out which element on the actual webpage might need to be added or removed or updated & we don’t write these concrete DOM updating instructions on our own as we would be doing that with Vanilla JS.
  1. **React project creation**
* Reference to get started with React: <https://github.com/facebook/create-react-app#creating-an-app>
* Steps

1. Download latest Node.js version: <https://nodejs.org/en> (Node JS is a technology that is not directly related to React, Node js is a runtime for JavaScript that allows us to run JS code outside the browser but React code is a JS code that runs on the browser.
2. To run following commands, we need Node JS runtime as pre-requisite

|  |
| --- |
| **// Way for creating new project**  npx create-react-app my-app  cd my-app  npm start  or  **// Way to import already created project**  npm install  npm start |

* **Newly created Project Structure**

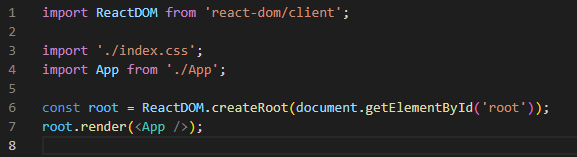


**React Project Structure Details**

**1. index.js**

* index.js file is the first code file which will be executed (<http://localhost:3000/>). It’s not exactly index.js file code but a transformed version of that code will be executed on the browser.
* Behind the scene, this **npm start** process not only start the development server & watch our code but also then takes the code, does the required transformation & deliver it to the browser.
* If we try to run index.js code directly, it won’t run in the browser (Invalid syntax)

**index.js**



**Step 1:** Here first we’re importing ReactDOM object from the react-dom 3rd party library which is one of our dependencies present in **package.json** which is downloaded & installed locally. Here we can see 2 React dependencies with different responsibilities i.e., react & react-dom (we can think of both together as React library)



**Step 2:** We call createRoot () method on ReactDOM object. This creates the main entry point, the main hook of the overall UI we’re about to build with React.

**ReactDOM.createRoot ()** tells React where this React application (UI we built) should be placed in the web page that will be loaded. This leads us to one other file i.e., index.html.

**2. index.html**

* **index.html** is the single HTML file which is in the end loaded by the browser. This is basically the only HTML file that is being used by this overall React application because it’s a so-called **single page application (SPA)** & all subsequent changes on the webpage will be handled by React.

**index.html** is the single HTML file or the entry point where the React-driven UI should be rendered in. We can see HTML document body structure (<div id=”root”></div>) i.e., div with id as root that doesn’t hold any content but that’s the div where we want to inject or render our React-driven UI.

Following code will be used for this purpose

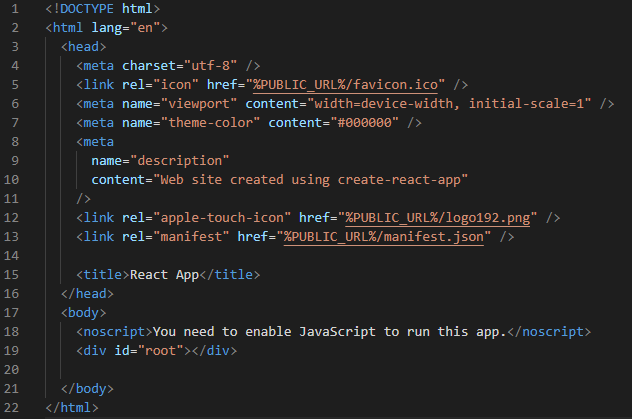




**root.render(<App />)** tells the React to render that div with id as root with the <App /> component present in App.js file.

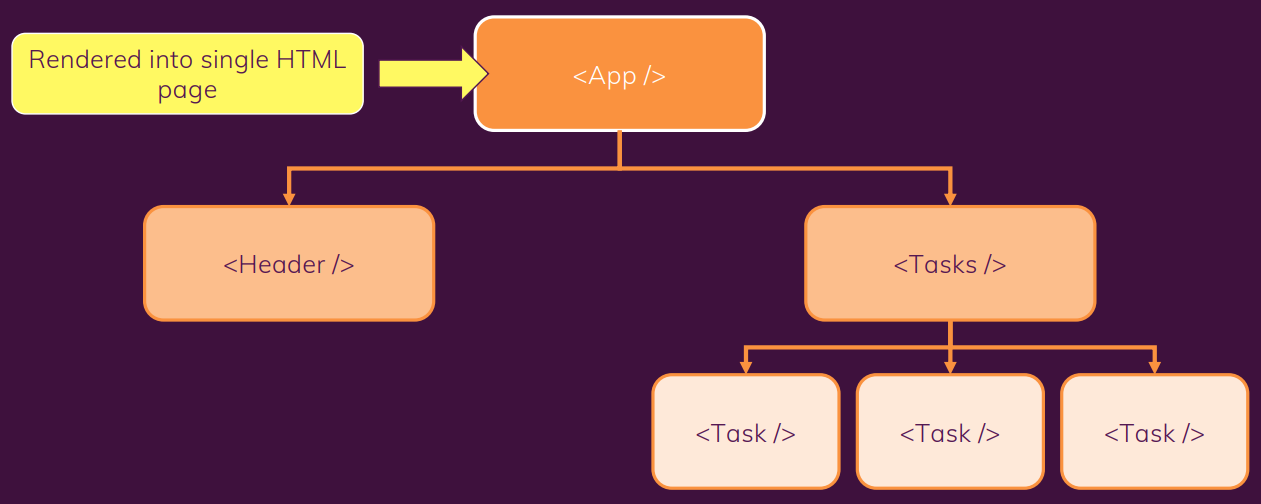
<App /> isn’t a regular JavaScript syntax, it’s **JSX syntax (JavaScript XML)**

**index.html** present in public folder



**3. App.js**

* This app component will be a special component called Root Component i.e., it’s the main component that will be render in our starting file index.js & all other components will be either nested inside App.js or nested inside of other components because ultimately with React, we build a Component Tree.



* 1. **Intro to JSX (JavaScript XML)**
* JSX is a special syntax introduced by the React team, & it works because of the transformation steps which are running behind the scenes.
* JSX is basically HTML code inside of JavaScript. JSX stands for JavaScript XML because HTML in the end is XML.
* We can see the transformed code by inspecting page source code in browser. It will contain entire react code along with our code.
* In order to output dynamic data, we put variables/constants/expressions inside curly braces {}
* JSX is stricter than HTML. We have to close tags like <br />. Our Component also can’t return multiple JSX tags. We have to wrap them into a shared parent like a <div>….</div> or empty <>….</> wrapper.

**Adding styles**

* In React, we specify a CSS class with “className”.

|  |
| --- |
| <img className="avatar" />  /\* In your CSS \*/  .avatar {  border-radius: 50%;  } |

**Displaying data (JSX with Curly Braces)**

* JSX lets us put markup into JavaScript. Curly braces {} let us “escape back” into JavaScript so that we can embed some variable from our code & display it to the user. For e.g.,

|  |
| --- |
| return (  <h1>  {user.name}  </h1>  ); |

* Curly braces let us bring JavaScript logic & variables into our markup.
* {{ and }} is not special syntax; it’s a JavaScript object tucked inside JSX curly braces.

**Conditional rendering**

* There are 3 ways for conditional rendering

1. **Using if / else statement** to conditionally include JSX
2. **Using ternary operator** {condition ? <A /> : <B />} mean if condition, render <A /> otherwise <B />.
3. **Using logical && syntax** {condition && <A /> } means if condition, render <A /> otherwise nothing.

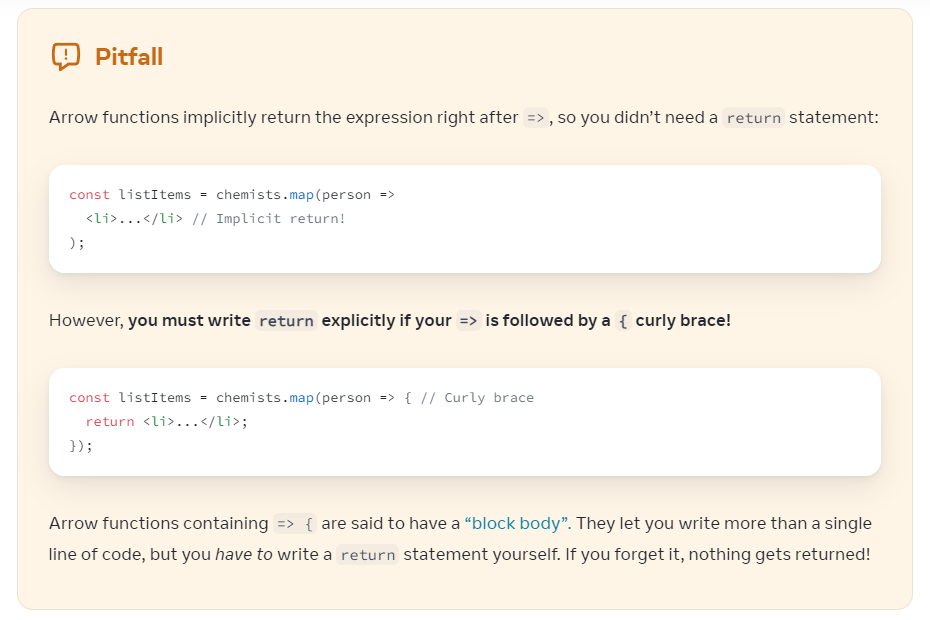
* We can put default statement in “let” variable & then update “let” variable based on the condition. Then embed the variable with curly braces in the returned JSX tree.
* In some situations, a component returns null if we don’t want to render anything at all. In practice, returning null from a component isn’t recommended because it might surprise a developer trying to render it. More often, we would conditionally include or exclude the component in the parent component’s JSX.

|  |  |  |
| --- | --- | --- |
| **Using if statement** | **Using ternay operator** | **Using logical && syntax** |
| let content;  if (isLoggedIn) {  content = <AdminPanel />;  } else {  content = <LoginForm />;  }  return (  <div>  {content}  </div>  ); | <div>  {isLoggedIn ? (  <AdminPanel />  ) : (  <LoginForm />  )}  </div> | <div>  {isLoggedIn && <AdminPanel />}  </div> |

**Rendering Lists**

* We will often want to display multiple similar components from a collection of data.
* In some case we need to show several instances of the same component using different data when building interfaces like from lists of comments to galleries of profile images. In these situations, we can store that data in JS objects & arrays and use methods like map() & filter () to render lists of components from them.
* We can use the JavaScript array methods to manipulate an array of data.
* Some commonly used array methods
* filter(), map(), forEach(), indexOf(), reduce(), reverse()

Notes:



**Keys**

* We need to give each array item a key – a string or a number that uniquely identifies it among other items in the array.
* JSX elements directly inside a map() call always need keys. Keys tell React which array item each component corresponds to, so that it can match them up later.
* Key becomes important if our array items can move (e.g., due to sorting), get inserted, or get deleted. A well – chosen key helps React infer what exactly has happened, & make the correct updates to the DOM tree.

**Where to get our key**

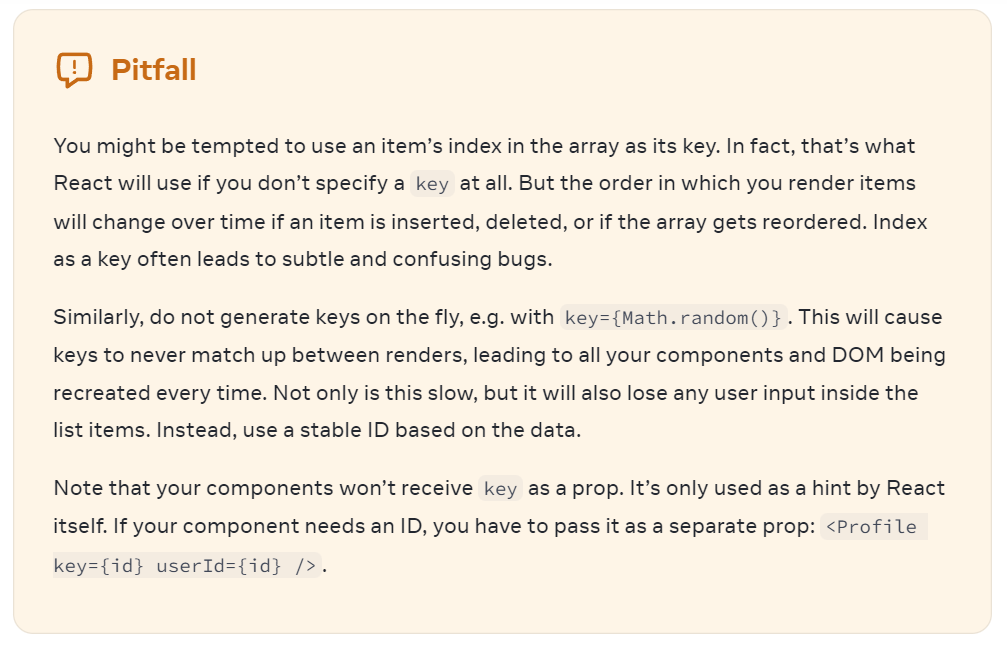
* Different sources of data provide different sources of keys:
* **Data from a database:** If our data is coming from a database, we can use the database keys/IDs, which are unique by nature.
* **Locally generated data:** If our data is generated & persisted locally (e.g., notes in a note – taking app), use an incrementing counter, crypto.randomUUID() or a package like uuid when creating items.

**Rules of keys**

* Keys must be unique among siblings. However, it’s okay to use the same keys for JSX nodes in different arrays.
* Keys must not change or that defeats their purpose! Don’t generate them while rendering.

**Why does React need keys?**

* Imagine that files on our desktop didn’t have names. Instead, we’d refer to them by their order – the 1st file, the 2nd file & so on. We could get used to it, but once we delete a file, it would get confusing. The 2nd file would become 1st, the 3rd file would be the 2nd file & so on.
* File names in a folder & JSX keys in an array serve a similar purpose. They let us uniquely identify an item between its siblings. A well – chosen key provides more information than the position within the array. Even if the position changes due to reordering, the key lets React identify the item throughout its lifetime.



**Keeping Components Pure**

* In Computer science (especially the world of functional programming), a pure function is a function with the following characteristics:
* It minds its own business i.e., It doesn’t change any objects or variables that existed before it was called.
* Same input, Same output i.e., Given the same input, a pure function should always return the same result.

Wrapper Components (Concept of composition)

* 1. **Custom Component**
* **Rule**: Components defined as Lowercase elements are built-in HTML elements while Components defined as Uppercase elements are custom component.
  1. **Props concept**
* React components use props to communicate with each other.
* Every parent component can pass some information to its child components by giving them props.
* Props might remind us of HTML attributes, but we can pass any JavaScript value through them including objects, arrays or functions.
  1. **Alternative Function syntax (Arrow function)**
* We can use arrow function instead of creating a function with “function” keyword.

|  |  |
| --- | --- |
| const ExpenseItem = (props) => {    return (        <Card>          <ExpenseDate date={props.date} />          <div>            <h2>{props.title}</h2>            <div>{props.amount}</div>          </div>        </Card>    ); | function ExpenseItem(props) {        return (        <Card>          <ExpenseDate date={props.date}/>          <div>            <h2>{props.title}</h2>            <div>{props.amount}</div>          </div>        </Card>      ); |

**2. React State & Working with Events**

**2.1 Listening to events & working with events Handlers**

* React lets us add event handlers to our JSX. Event handlers are our own functions that will be triggered in response to interactions like clicking, hovering, focusing form inputs, & so on.
* To add an event handler, we will first define a function & then pass it as a prop to the appropriate JSX tag.

e.g., We can define the clickHandler in 3 ways (All of these styles are equivalient.):

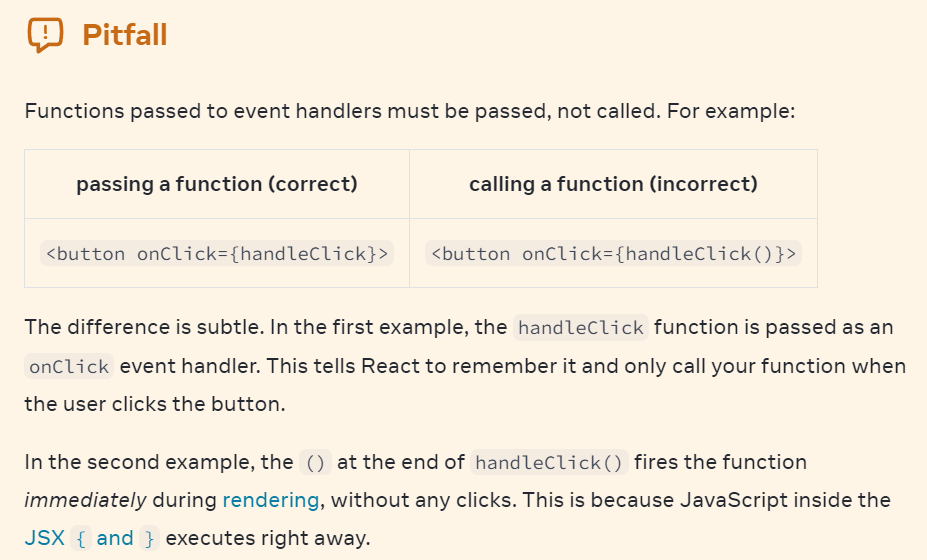
1. Normal function definition
2. Using Inline function definition
3. Using Arrow function definition

* By convention, it’s common to name event handlers as “**handle**” followed by the event name or Event name followed by “**handler**”.

e.g., onClick = { handleClick } **or** onClick = { clickHandler }

|  |
| --- |
| // Normal function definition  export default function ClickButton() {    const clickHandler = () => {      alert("Button has been clicked!!");    };    return (      <div>        <button onClick={clickHandler}>Click me!!</button>      </div>    );  } |
| // Using Inline function definition  export default function ClickButton() {    return (      <div>        <button          onClick={function handleClick() {            alert("Online Button has been click!!");          }}        >          Click me Inline!!        </button>      </div>    );  } |
| // Using Arrow function definition  export default function ClickButton() {    return (      <div>        <button          onClick={() => {            alert("Arrow function Button has been click!!");          }}        >          Click me arrow function        </button>      </div>    );  } |

**Notes:**



**Passing event handlers as props**

* Often, **we’ll want the parent component to specify a child’s event handler.**
* E.g., Consider buttons: depending on where we’re using a Button component, we might want to execute a different function – perhaps one plays a movie & another uploads an image.
* To do this, we pass the event handler as a prop to the component from its parent.
* Built – in components like <button> & <div> only support browser event names like onClick etc. However, when we’re building our own components, we can name our event handler props anyway that we like
* By convention, event handler props should start with “on”, followed by a capital letter like **onSmash, onPlayMovie, onUploadImage** etc.



* It’s common for components like buttons to contain styling but not specify behavior. Instead, components like **PlayButton** & **UploadButton** will pass event handlers down.

**Event Propagation**

* Event handlers will also catch events from any children our main component have. We say that an event “bubbles” or “propagates” up the tree; it starts with where the event happened & then goes up the tree.

e.g.,





* If we click on either button, it’s onClick will run first, followed by the parent <div>’s onClick & if we click the toolbar itself, only the parent <div>’s onClick will run.

**Stopping Propagation**

* Event handlers receive an event object as their only argument.
* By convention, it’s usually called “e”, which stands for “event”. We can use this object to read information about the event.
* That event object also lets us stop the propagation. If we want to prevent an event from reaching parent components, we need to call **e.stopPropagation()**





* When we click on a **button** (**PlayMovie**):

1. React calls the clickHandler passed to <button>.
2. **clickHandler** does the following:

* Calls event.stopPropagation(), preventing the event from bubbling or propagating up.
* Calls the onSmash() function, which is a prop passed from the StopEventPropagation component

1. onSmash() function is defined in the StopEventPropagation component, that displays the Button’s content.
2. Since the propagation was stopped, the parent <div>’s onClick handler doesn’t run.

* As a result of e.stopPropagation(), clicking on the buttons now only show a single alert(from the <button>) rather than the two of them (from the <button> & the parent toolbar <div>).

**Passing Handlers as alternative to propagation**

* We can add more code to the handler before calling the parent onClick event handler (see above e.g.) 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 we can clearly follow the whole chain of code that executes as a result of some event.

**Preventing default behavior**

* Some browser events have default behavior associated with them. For e.g., a <form> submit event, which happens when a button inside of it is clicked, will reload the whole page by default.
* We can call **event.preventDefault()** on the event object to stop this from beginning.
  1. Component function Execution
  2. Working with “State” & “useState” Hook
  3. Ways to update state
  4. Adding Form inputs, Listening to User Input & Handling form submission
  5. Working with Multiple states Vs Using One state
  6. Updating state that depends on the previous state
  7. Adding 2-way binding
  8. Child to Parent Component Communication (Bottom-up), Lifting the State Up
  9. Controlled Vs Uncontrolled Components, Stateless Vs Stateful Components