# Jonas Schmedtmann | The Ultimate React Course 2023: React, Redux, & More

Helpful resources for part 1 of the course:

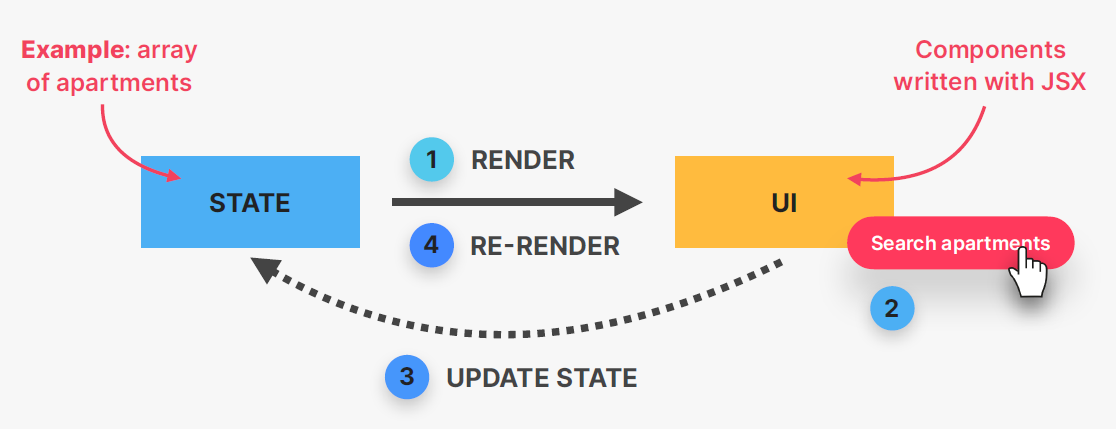
* Adding react URL to an HTML document
  + <https://gist.githubusercontent.com/gaearon/0275b1e1518599bbeafcde4722e79ed1/raw/db72dcbf3384ee1708c4a07d3be79860db04bff0/example.html>
* React Documentation <https://react.dev/?ref=jonas.io>
* <https://create-react-app.dev/?ref=jonas.io> – this is how we will set up our first app
* Vite: Getting Started – for real-world apps
  + <https://vitejs.dev/guide/?ref=jonas.io>

# Why do front-end frameworks exist? Why not just use Vanilla JS?

1. Keeping a user interface in sync with data is **really difficult** and a **lot of nightmarish work**
   1. Front end frameworks solve this problem and take hard work away from developers 🎉
2. They enforce a “**correct**” way of structuring and writing code (therefore contributing to solving the problem of “spaghetti code” 🍝)
3. They give developers and teams a **consistent** way of building front-end apps

# What is React?

* A JavaScript library for building user interfaces
  + A.k.a., extremely popular declarative, component-based state-driven **JavaScript library for building user interfaces**, created by an engineer at Facebook in like 2011.
* React is based on **components**
  + Components are the **building blocks** of user interfaces in React
  + We build complex UIs by **building and combining multiple components**
* React is **declarative**
  + We **describe** how components look like and how they work using a **declarative syntax called JSX**
  + **Declarative**: telling React what a component should look like, **based on current data/state**
  + React is **abstraction** away from the DOM: we **never touch the DOM**
  + **JSX**: a syntax that **combines** HTML, CSS, and JavaScript as well as referencing **other components**
* React is **state-driven**
  + React ***reacts*** to state changes by re-rendering the UI



* React is a JavaScript library
  + Common question: Is React a library or a framework?
    - It’s a library because React is only the “view” layer. We need to pick multiple external libraries to build a complete application
    - There are complete frameworks built on top of React
      * E.g., Next.js and Remix
* React is **extremely popular**
  + Many large companies have adopted React
    - PayPal, Tesla, Netflix, IMDb, Airbnb, and Dropbox
  + Huge job market with high demand for React developers
  + Large and vibrant React developer community
  + Gigantic third-party library ecosystem
* React was **created by Facebook**
  + React was created in **2011** by Jordan Walke, an engineer working at Facebook at the time
  + React was open-sourced in 2013, and has since then completely transformed front-end web development

# Summary – React is good at two things:

* Rendering components on a webpage (UI) based on their current state
* Keeping the UI in sync with state, by re-rendering (*reacting*) when state changes

Code snippets

* Pieces of predefined code we can use to speed up development

Setting Up a New React Project: Options

## The **TWO OPTIONS** FOR SETTING UP A REACT PROJECT

* **Create-React-App** – use for tutorials or experiments, but not for real-world apps
  + Complete “**starter kit**” for React applications
  + Everything is **already configured**: ESLint, Prettier, Jest, etc.
  + 👎🏻 Uses **slow and outdated** technologies (i.e., webpack)
* **Vite** – use for modern **real-world apps**
  + **Modern build tool** that contains a **template** for setting up React applications
  + 👎🏻 Need to manually set up ESLint (and others)
  + Extremely fast hot module replacement (HMR) and bundling

What about React **frameworks**?

* The React team now advises to use a “**React framework**” for new projects
  + Such as Next.js or Remix
* Many people think that this is not the best idea: “**vanilla**” **React apps are important too!**
* This only makes sense for building actual products, **not learning React**
* Of course, you still need to learn React itself
* Don’t worry about this recommendation for now

# Setting Up a Project with Create-React-App

* Open up your **Command Prompt/Terminal**
* **cd** into your Desktop folder (or wherever you store the project)
* run the command: npx create-react-app@5 pizza-menu
  + downloads thousands of packages, takes some time

A screenshot of a computer program

Description automatically generated

* Once downloads complete, close Terminal
* (rename the folder 03-pizza-menu to be consistent with project folders)
* Open project folder in VS Code
* You will see that a project folder with all necessary packages has been created
  + Look under **node\_modules** for the react and react-dom **packages**
  + The **public** folder contains all the **assets** that end up in the final application, as well as the **index.html** file
    - create-react-app automatically creates these files for us. In the index.html file, you’ll see the div#root already there. In index.js, a constant is created from this div#root and it renders the app inside this div for us.
* To start the app, look in package.json at the npm scripts key “start”
* Open a new Terminal in VS Code
* Run the command: npm run start
  + Or just npm start
  + This will start the application and open it in a browser window for you. Tab title is set to “React App” and has the little atomic logo as a favicon.

WORKING WITH COMPONENTS, PROPS, AND JSX

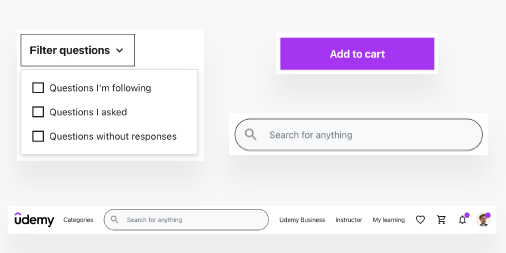
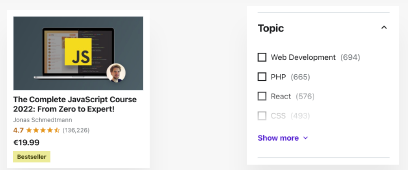
* Core concepts: components, props, JSX
* Creating and reusing components
* Rendering lists
* Conditional rendering
* Start writing code on your own

# Before coding, some advice for debugging and dealing with errors:

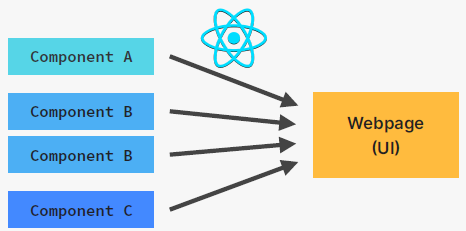
* Application not updating on save?
  + Check to make sure the app is actually running
  + Stop and restart
    - Control + C to stop
    - **npm start**
  + do a hard reload in browser
  + have the browser console open so that you immediately see error messages and stuff
  + copy and paste the error message into Google if needed
  + always work with ESLint
* Can’t figure out where the bug is in your code? Look at the final version

# Components as Building Blocks

* React apps are entirely made out of **components**
  + Building blocks of user interfaces in React

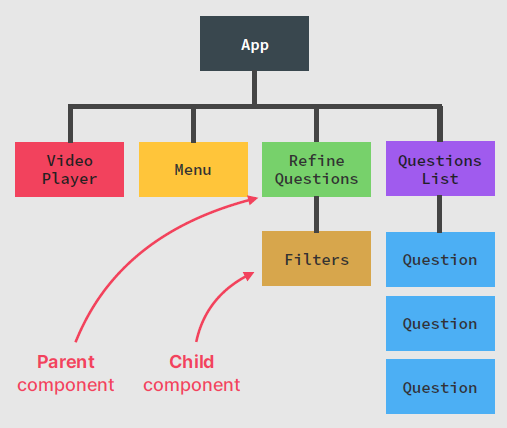
 

* Piece of UI that has its own **data, logic,** and **appearance** (*how it works and looks*)
* We build complex UIs by **building multiple components** and **combining** them

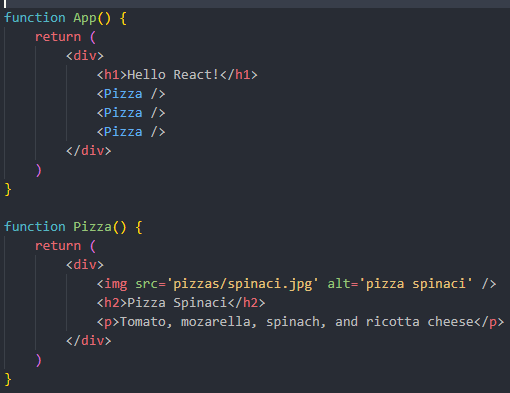
* Components can be **reused**, **nested** inside each other, and **pass data** between them

# Component Trees



# Creating and Reusing a Component

* Each component can return exactly *one* element
  + If you want to return multiple elements, wrap them in a <div> or React fragment <></>
* All your components are declared in the top level of your app (<App />)
* Components can be nested in other components

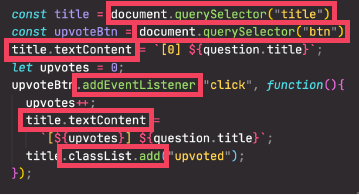


# What is JSX?

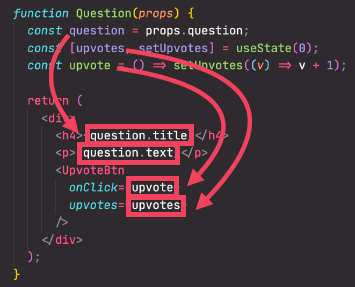
* Declarative syntax to **describe** what components **look like** and **how they work**
* Components must **return** a block of JSX
* Extension of JavaScript that allows us to **embed JavaScript, CSS, and React components into HTML**
* Each JSX element is **converted** to a React.createElement function call
* We could use React **without JSX** but we don’t

# JSX is Declarative

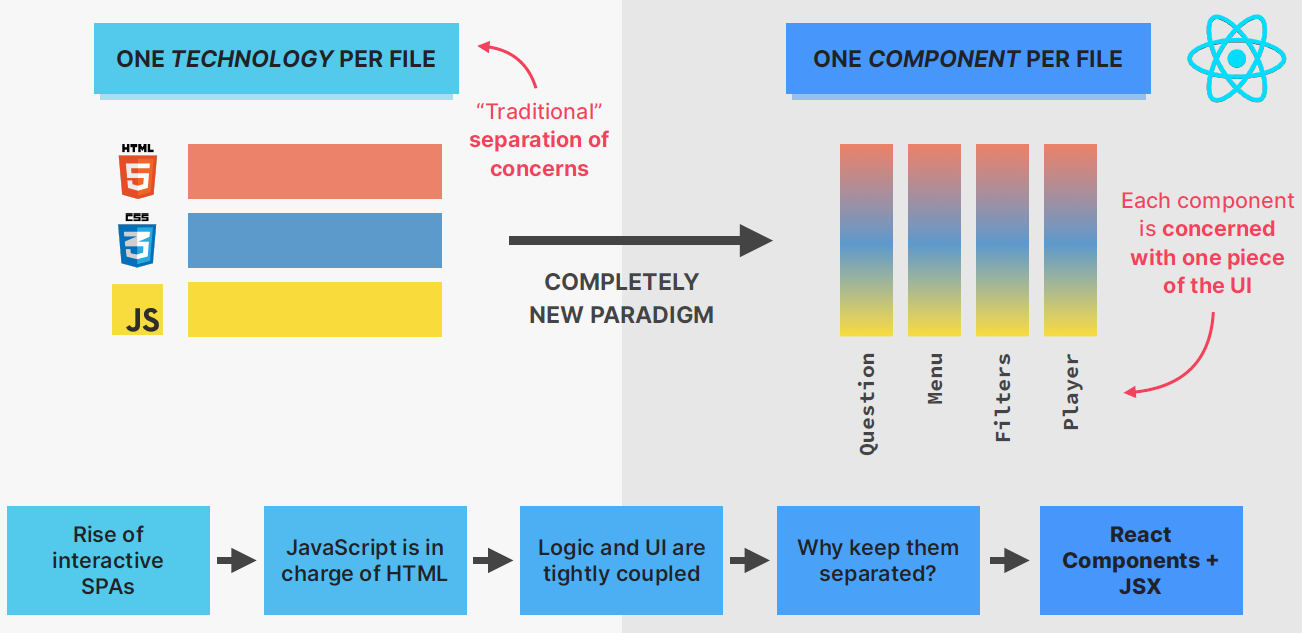
* Imperative – “*how* to do things”; not recommended for complex apps
  + Manual DOM element selections and DOM traversing
  + Step-by-step DOM mutations until we reach the desired UI



* Declarative – “*what* we want.”
  + Describe what UI should look like using JSX, **based on current data**
  + React is an **abstraction** away from DOM: **we never touch the DOM**
  + Instead, we think of the UI as a **reflection of the current data**



# Separation of Concerns



# Props, Immutability, and One-Way Data Flow

## Props

* Used to pass data from **parent components** to **child components** (down the component tree)
* Essential tool to **configure** and **customize** components (like function parameters)
* With props, parent components **control** how child components look and work
* **Anything** can be passed as props
  + single values, arrays, objects, functions, other components

## Props are READ-ONLY

* *props* are data coming from the **outside**, and can **only** be updated by the **parent component**
* *props* are read-only, they are **immutable** – this is one of React’s strict rules
* if you need to mutate props, you actually need **state**
  + **why tho?**
  + Mutating props would affect parent, creating side effects (not pure)
  + Components have to be **pure functions** in terms of props and state
  + This allows React to optimize apps, avoid bugs, make apps predictable
* **state** is internal data that can be updated by the **component’s** logic

## One-Way Data Flow

* makes applications more predictable and easier to understand
* makes applications easier to debug, as we have more control over the data
* more performant
* Angular has **two-way** data flow

# Rules of JSX

## General JSX Rules

* JSX works essentially like HTML, but we can enter “**JavaScript mode**” by using curly brackets {} (for text or attributes)
* We can place **JavaScript expressions** inside **{}**
  + E.g., reference variables, create arrays or objects, [].map(), ternary operator
* Statements are **not allowed** (if/else, for, switch)
* JSX produces a JavaScript **expression**



* A piece of JSX can only have **one root element.** If you need more, use <React.fragment>…</React.fragment> or the shortened <>…</>

## Differences between JSX and HTML

* className instead of HTML’s class
* htmlFor instead of HTML’s for
* Every tag needs to be **closed**.
* All event handlers and other properties need to be **camelCased**
  + **Exception:** aria-\* and data-\* are written with dashes like in HTML
* CSS inline styles are written like this: {{<style>}} (to reference a variable, and *then* an object)
* CSS property names are also **camelCased**
* Comments need to be in {} (because they are JS)

## Rendering Lists

* For when we have an array of objects with data and we need to create a component for each one and include some or all of its associated data

Figure A – the object array containing all the pizzas

A computer screen shot of a program code

Description automatically generated

Figure B

A computer screen shot of a computer code

Description automatically generated

* The .**map**() method is most commonly used to render lists like this in React
  + This is a regular ES6 JavaScript concept
* We can map over each object in our data array (**pizzaData**) and create a Pizza component for each one
  + props dynamically passed in
* dynamically passing in props like that can get really complicated if you’ve got a lot of data
* Usually, we pass in the entire pizza object into the more specific (instance) component

Figure C

A computer code with colorful text

Description automatically generated with medium confidence

* And make sure to adapt the Pizza func accordingly

A screen shot of a computer code

Description automatically generated

* When we form a list with the .map() method (in React), every item that gets rendered needs a unique ***key*** property
  + The key prop is internal to React; needed for performance optimizations

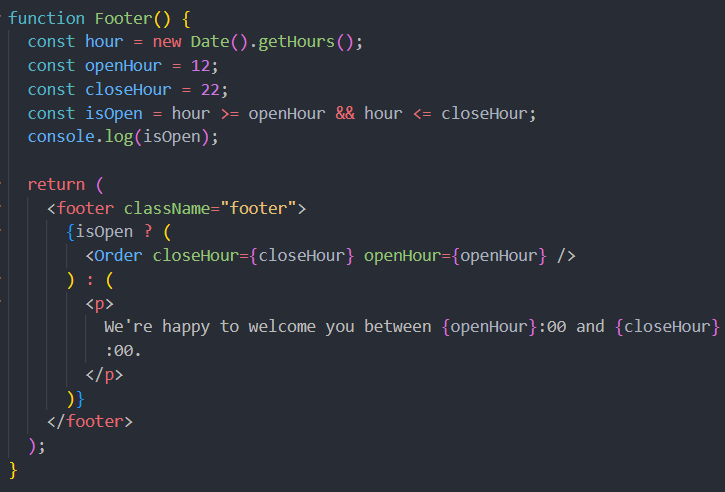
# Conditional Rendering with &&

* (still using the pizza example project) now we want to render a blurb in the footer based on whether the restaurant is currently open or not
  + i.e., rendering UI based on a condition
* recall how the && operator works
  + both (or all) conditions must evaluate to true

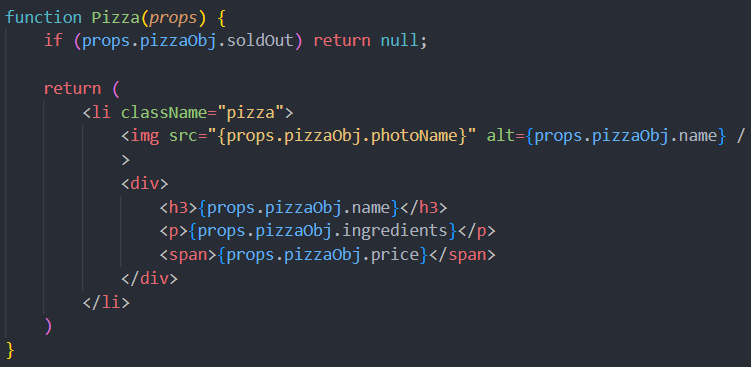
A computer code with colorful text

Description automatically generated

# Conditional Rendering with the Ternary Operator



# Conditional Rendering with Multiple Returns

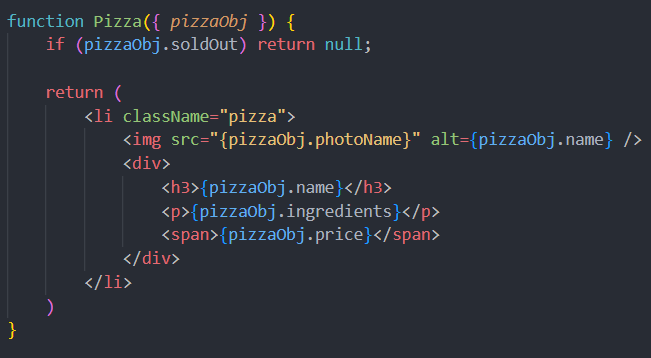


# Extracting JSX into a New Component

* When the JSX in a component is getting too clunky, we can extract it into its own component
* If the JSX depends on some value in the parent component, we simply pass it in as a prop

# Destructuring Props

* Destructuring props allows for cleaner code and fewer steps
* So instead of receiving just the *props* object, you can get more immediately specific by de-structuring going in



STATE, EVENTS, AND FORMS: INTERACTIVE COMPONENTS

# What is *state* in React?

* **State** is the most important concept in React
* It’s kind of the whole point

# What you need to learn about state:

* **What is it** and **why** do we need it?
* How to use state in **practice**?
  + useState
  + useReducer
  + Context API
* **Thinking** about state
  + when to use state
  + where to place state
  + types of state

# State

* Data that a component **can hold over time**, necessary for information that it needs to **remember** throughout the app’s lifecycle
* “**Component’s memory**” 🧠

A diagram of components

Description automatically generated A screenshot of a computer

Description automatically generated

* “**State variable**” / “**piece of state**”: a single variable in a component (**component state**)
* Updating a **component state** triggers React to **re-render the component**

A diagram of a process

Description automatically generated

A diagram of a process

Description automatically generated

---------------------------------------------------------------------------------------------------------------

A computer screen shot of a computer process

Description automatically generated

* React is called “React” because it **REACTS** to state changes by re-rendering the UI

# One Component, One **State**

* Each component has and manages **its own state**, no matter how many times we render the same component
* UI as a function of state
  + With state, we view UI as a **reflection of data changing over time**
  + We ***describe* that reflection** of data using state, event handlers, and JSX
    - React is *declarative*

# Practical guidelines about state

* Use a state variable for any data that the component should keep track of (“remember”) over time.
  + **This is data that will change at some point.** In vanilla JS, that’s a let variable, or an [] or {}.
* Whenever you want something in the component to be **dynamic**, create a piece of state related to that “thing”, and update the state when the “thing” should change (aka “be dynamic”)
  + Example: A modal window can be open or closed. So we create a state variable isOpen that tracks whether the modal is open or not. On isOpen = true we display the window, whereas on isOpen = false we hide it.
* If you want to change the way a component looks, or the data it displays, **update its state**. This usually happens in an **event handler** function.
* When building a component, imagine its view as a **reflection of state changing over time**
* For data that should not trigger component re-renders, **don’t use state**. Use a regular variable instead. This is a common **beginner mistake**.

# State vs Props

## State

* **Internal** data, owned by component
* Component “memory”
* Can be updated by the component itself
* Updating state causes component to re-render
* Used to make components interactive

## Props

* **External** data, owned by parent component
* Similar to function parameters
* Read-only (cannot be modified by the component receiving them)
* **Receiving new props causes component to re-render.** Usually when parent’s state has been updated.
* Used by parent to configure child component (“settings”)

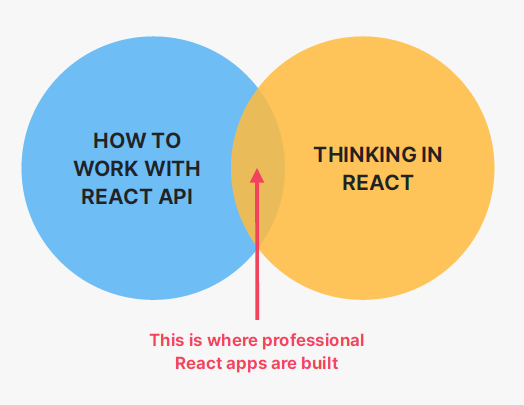
Packing list project breakdown visualization

[paste in scanned document]

Section 7 – Thinking in React: State Management

# What is “Thinking in React?” 🤨

* Thinking in React is a core skill
  + “***React Mindset***”
  + Thinking about components, state, data flow, effects, etc.
  + Thinking in **state transitions**, not element mutations



# The “Thinking in React” Process

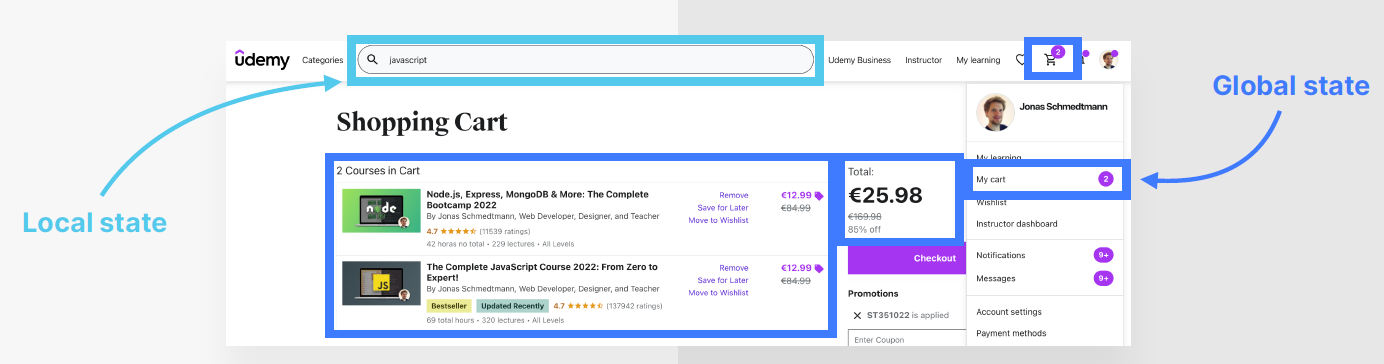
* The process is not a rigid one
* Break the desired UI into **components** and establish the **component tree**
* Build a static version in React (without state)
* Think about **state**:
  + When to use state
  + Types of state: local vs. global
  + Where to place each piece of state
* Establish **data flow**:
  + One-way data flow
  + Child-to-parent communication
  + Accessing global state
* When you know how to think in React, you will then know exactly how to:
  + Break up a design into components
  + Make some components reusable
  + Assemble UI from reusable components
  + Determine what pieces of state are needed for interactivity
  + Determine what types of state you can/should use
  + Make data flow through the app

# Fundamentals of State Management

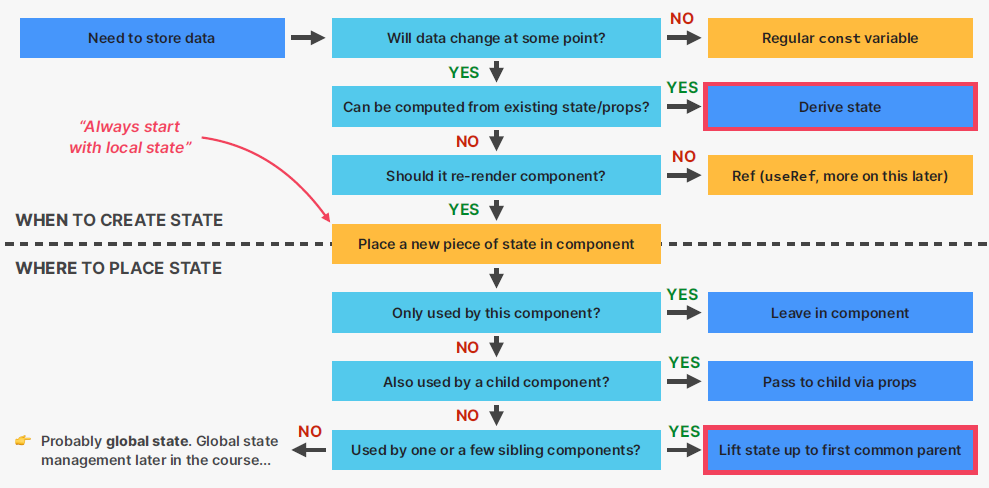
* State management: deciding **when** to create pieces of state, what **types** of state are necessary, **where** to place each piece of state, and how data **flows** through the app
  + Giving each piece of state a home 🏠

## Local vs. Global State

* Local state
  + State needed **only by one or a few components**
  + State that is defined in a component and **only *that* component, and child components** have access to it (by passing via props)
  + **We should always start with local state**
* Global state
  + State that **many components** might need
  + **Shared** state that is accessible to **every component** in the entire application



# State: When and Where?



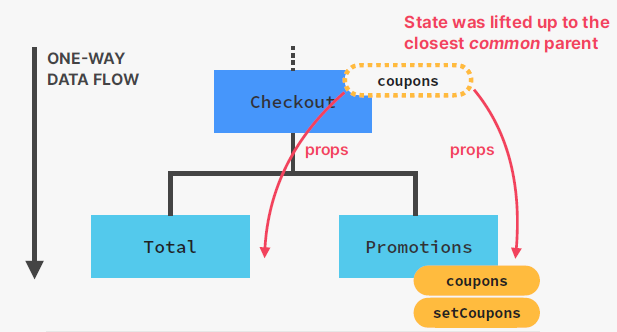
# Thinking About State and Lifting State Up

* Problem! Sharing with the sibling component (lol much like actual sibling rivalries)

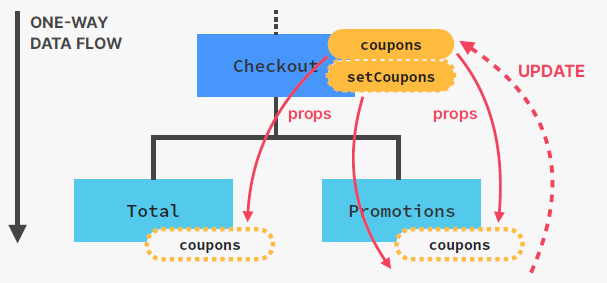
A diagram of a flowchart

Description automatically generated

* Data flows one-way down the component tree
  + Not sideways to siblings
* So how do we share state with **other components**?
  + We “lift” the state up to the closest *common* parent
* By lifting state up, we have successfully shared one piece of state with multiple components in different positions in the component tree



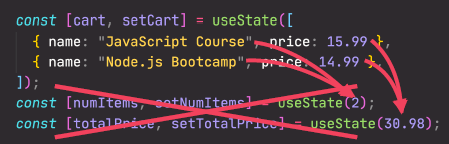
# Child-to-Parent communication



* If data flows from parent to children, how can Promotions (child) update state in Checkout (parent)?
  + **Child-to-parent communication (inverse data flow)**: child updating parent state (data “flowing” **up**)

# Derived State

Deriving State



* Here we have three separate pieces of state, event though numItems and totalPrice depend on cart 👎
* Need to keep them in sync (update together) 👎
* 3 state updates will cause 3 re-renders… is that really necessary? 🤔

A close up of a sign

Description automatically generated

Derived state: state that is computed from an existing piece of state or from props

A screen shot of a computer program

Description automatically generated

* Just regular variables, no useState
* cart state is the single source of truth for this related data
* Works because re-rendering component will automatically re-calculate derived state

# The children Prop – Making a Reusable Button

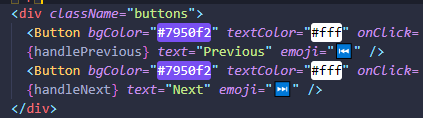
A close-up of a button

Description automatically generated

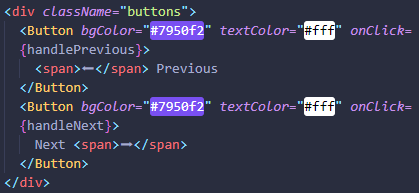
* The **children** prop allows us to pass JSX into an element (besides regular props)
* Essential tool to make reusable and configurable components (especially component content)
* Really useful for generic components that don’t know their content before being used (e.g. modal)
* Notice how the <Button /> component (in 04-steps) is taking in a lot of props. We could add more, but there is another option…
* (And besides that, notice how we want our emoji to be on the left of the text in one instance, but in another we want it to be on the right of the text… set up with the passed in text and emoji props, this is limiting…)

 I want the emoji on the right

* + JSX, like html, can **utilize closing tags**
  + E.g. add a </Button> and remove the text and emoji props.
  + Inside the <Button><Button />:
    - Set up your emoji in a <span> and hard coded “Previous”/”Next”
      * Instead of this:



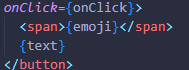
* + - * Do this:



* + Upon saving the code (still in 04-steps), notice the content of the buttons goes away
    - We need to pass in the *children* of the <Button></Button>
      * ***children***= contents inside; everything between the opening and closing tags
        + in this case, our <span> containing the emoji and accompanying text
  + instead of passing in the *text* and *emoji* props, pass in the *children* props







A computer screen shot of a code

Description automatically generated with medium confidence

PART 02 – INTERMEDIATE REACT

THINKING IN REACT: COMPONENTS, COMPOSITION, AND REUSABILITY

# Lecture: How to Split a UI into Components

## Components size matters

* generally, we need to go for the right balance between too specific and too broad.
* HUGE: writing in a way that produces just one big ass component comes with serious drawbacks
  + too many **responsibilities**
  + might need too many **props**
  + hard to **reuse**
  + **complex** code, hard to understand
* SMALL: keeping components wayyy small, a little too atomic
  + you'll end up with hundreds of mini-components
  + confusing codebase
  + too **abstracted**
    - abstraction – creating something new to hide the implementation details of that thing

## How to **Split** a UI into Components

* the 4 criteria for splitting a UI into components:
  + logical separation of content/layout
  + reusability
  + responsibilities/complexity
  + personal coding style

## Framework: When to Create a New Component

Tip: when in doubt, start with a relatively large component, then split it into smaller components as it becomes necessary

* Logical separation of content/layout
  + Does the component contain pieces of content or layout that **don’t belong together**?
* Reusability
  + Is it possible to reuse part of the component?
  + Do you **want** or **need** to reuse it?
* Responsibilities/complexity
  + Is the component doing too **many different things**?
  + Does the component rely on too **many props**?
  + Does the component have too **many pieces of state** and/or effects?
  + Is the code, including JSX, too **complex**/**confusing**?

## Some More General **Guidelines**

* Be aware that creating a new component **creates a new abstraction**.
  + Abstractions have a cost, because **more abstractions require more mental energy** to switch back and forth between components.
  + So try not to create new components too early
* Name a component according to **what it does** or **what it displays**. Don’t be afraid to use long component names.
* Never declare a new component **inside another component**!
* **Co-locate related components inside the same file**. Don’t separate components into different files too early.
* It’s completely normal that an app has components of **many different sizes**, including very small and huge ones

## Any App has Components of Different Sizes and Reusability

* Components with all different sizes
  + Different degrees of size, reusability, responsibility, and complexity
* Small
  + **Some very small components are necessary**
  + Highly reusable
  + Very low complexity
* Huge
  + **Most apps will have a few huge components**
  + Not meant to be reused (not a problem)

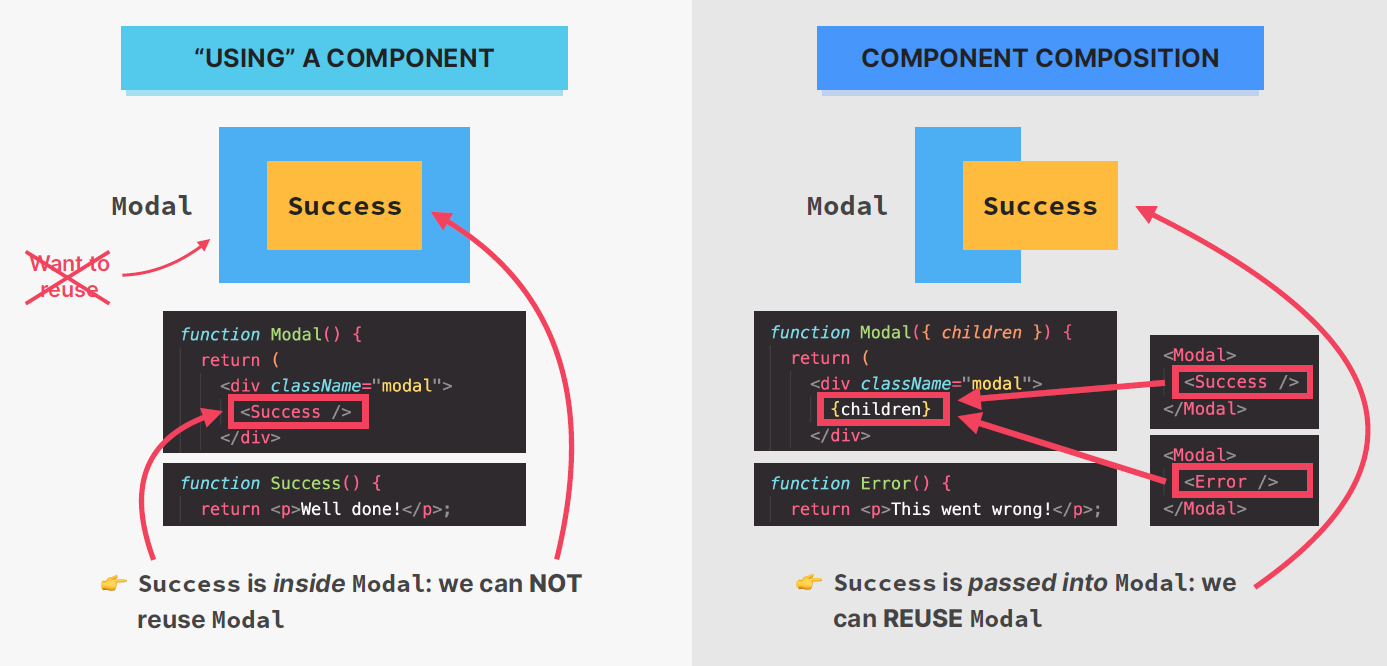
# Lecture: Component Categories

## Component categories

* Most of your components will **naturally fall into one of three categories**:
  + Stateless/presentational components
    - No state
    - Can receive props and simply *present* received data or other content
    - Usually **small and reusable**
  + Stateful components
    - Have state
    - Can still be reusable
  + Structural components
    - “**pages**”, “**layouts**”, or “**screens**” of the app
    - Result of **composition**
    - Can be **huge and non-reusable** (but don’t have to)

# Lecture: Component Composition

* Component composition: combining different components using the **children** prop (or explicitly defined props)
  + Components don’t need to know their children in advance
* With component composition, we can:
  + Create highly **reusable and flexible** components
  + Avoid dealing with prop drilling (great for layouts)
    - Prop drilling – the pain in the ass that comes with having to pass props down the tree through so many components that don’t even use them (see previous video lecture)



HOW REACT WORKS BEHIND THE SCENES

# Lecture: Components, Instances, and Elements

**Component**

* **description** of a piece of UI
* a function that **returns React elements** (element tree), usually written as JSX
* “blueprint” or “template”

**Component Instance**

* Instances are created when we “use” components
* React internally calls Tab()
* actual “**physical**” **manifestation** of a component
* has its own state and props
* has a **lifecycle** (can be born, lives, dies)

React Element

* JSX is converted to React.createElement() **function calls**
* A React element is the **result of these function calls**
* Information necessary to create **DOM elements**

DOM Element (HTML)

* Actual **visual representation** of the component instance in the browser

# Overview: How Components are Displayed On the Screen

* In React, rendering is NOT updating the DOM or displaying elements on the screen. Rendering only happens **internally** inside React, it does not **produce visual changes**

A diagram of a diagram

Description automatically generated

# How Renders are Triggered

* The two situations that trigger renders:
  + **Initial render** of the app
  + **State is updated** in one or more component instances (**re-render**)
* The render process is triggered for the **entire application**
* In practice, it looks like React only re-renders the component where the state update happens, but that’s not how it **works behind the scenes**.
* Renders are **not** triggered immediately, but **scheduled** for when the JS engine has some “free time”. There is also batching of multiple setState calls in event handlers.

# Lecture: How Rendering Works – The Render Phase

Review: the mechanics of state in React

A diagram of a process

Description automatically generated

Ok so some stuff we’ve learned already is technically not true in a very technical techy sense

NOT TRUE #1: rendering is updating the screen/DOM

NOT TRUE #2: React completely discards old view (DOM) on re-render

# The Render Phase

A blue rectangles with black text

Description automatically generated

## The Virtual DOM (React Element Tree)

* *Virtual DOM*: Tree of all React elements created from all instances in the component tree
* Cheap and fast to create multiple trees
* Nothing to do with “shadow DOM”
* Rendering a component will **cause all of its child components to be rendered as well** (no matter if props changed or not)
  + This is necessary because React doesn’t know whether children will be affected

## What is **Reconciliation** and Why Do We Need It?

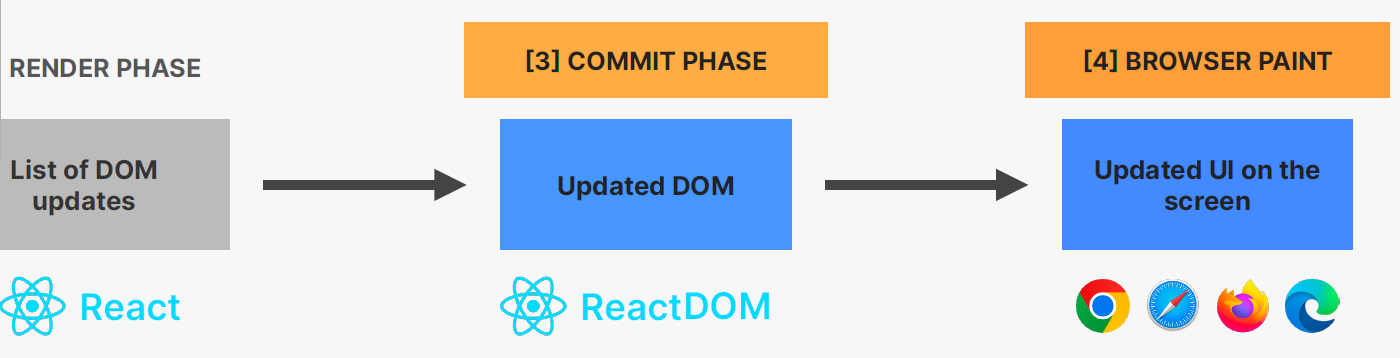
* Why not update the entire DOM whenever state changes somewhere in the app? 🤔
  + Because.
  + That would be inefficient and wasteful:
    - Writing to the DOM is (relatively) slow
    - Usually only a **small part of the DOM** needs to be updated
  + React **reuses** as much of the existing DOM as possible
    - Ho0oOw? You ask.
    - **Reconciliation**: Deciding which DOM elements actually need to be inserted, deleted, or updated in order to reflect the latest state changes

## The Reconciler: Fiber

* *Fiber tree*: internal tree that has a “fiber” for each component instance and DOM element
* Fibers are **NOT** re-created on every render
* Work can be done **asynchronously**
* Rendering process can be split into chunks, tasks can be prioritized, and work can be **paused**, **reused**, or **thrown away**
  + Enables **concurrent features** like suspense or transitions
  + Long renders **won’t block** JS engine

# Lecture: How Rendering Works – The Commit Phase

## The **Commit** Phase and Browser **Paint**



* **React writes to the DOM**: insertions, deletions, and updates (list of DOM updates are “flushed” to the DOM)
* **Committing is synchronous**: DOM is updated in one go, it can’t be interrupted. This is necessary so that the DOM never shows partial results, ensuring a consistent UI (in sync with state at all times)
* After the commit phase completes, the workInProgress fiber tree becomes the current tree for the **next render cycle**
* So at the list of DOM updates point (see figure above), React does **not** touch the DOM. React **only renders**.
  + It doesn’t know where the render result will go
  + React can be used on different platforms (“hosts”)
* At the updated DOM point: There are many “renderers” you can use, like ReactDOM, React Native, Remotion, etc.
  + “renderers” do not actually render; they ***commit*** the result of render phase

# Lecture: How Diffing Works

* Diffing uses 2 fundamental assumptions (rules):
  + Two elements of different types will **produce different trees**
  + Elements with a stable key prop **stay the same across renders**
* This allows React to go from **1,000,000,000** [O(n^3)] to **1000** [O(n)] operations per 1000 elements
* Same position, **different** element
  + React assumes **entire sub-tree is no longer valid**
  + Old components are destroyed and removed from DOM, **including state**
  + Tree might be rebuilt if children stayed the same (**state is reset**)
* Same position, **same** element
  + Element will be kept (as well as child elements), including **state**
  + New props/attributes are passed if they changed between renders
  + Sometimes this is **not** what we want… then we can use the key prop

# Lecture: the Key Prop

* Special prop that we use to tell the diffing algorithm that an element is **unique**
* Allows React to **distinguish** between multiple instances of the same component type
* When a key **stays the same across renders**, the element will be kept in the DOM (*even if the position in the tree changes*)
  + Using keys in lists
* When a key **changes between renders**, the element will be destroyed and a new one will be created (*even if the position in the tree is the same as before*)
  + Using keys to reset state

# Lecture – Rules for Render Logic: Pure Components

## THE **TWO TYPES** OF LOGIC IN REACT

1. render logic
   1. code that lives at the top level of the component function
   2. participates in describing how the component view looks like
   3. executed every time the component renders
2. event handler functions
   1. executed as a **consequence of the event** that the handler is listening for (change event in this example)
   2. code that actually **does things**: update state, perform an HTTP request, read an input field, navigate to another page, etc.

## REFRESHER: FUNCTIONAL PROGRAMMING PRINCIPLES

* **Side effect**: dependency on or modification of any data outside the function scope.
  + “*interaction with the outside world.*”
  + Examples: mutating external variables, HTTP requests, writing to DOM.
  + Note: **side effects are not bad**! A program can only be useful if it has some interaction with the outside world
* **Pure function**: a function that has **no** side effects
  + Does **not** change any variables outside its scope
  + Given the **same input**, a pure function always returns the **same output**

## RULES FOR RENDER LOGIC

* Components must be pure when it comes to render logic: given the same props (input), a component instance should always return the same JSX (output)
* Render logic must produce no side effects: no interaction with the “outside world” is allowed. So, in render logic:
  + Do NOT perform **network requests** (API calls)
  + Do NOT start **timers**
  + Do NOT directly **use the DOM API**
  + Do NOT **mutate objects or variables** outside of the function scope
  + Do NOT **update state (or refs)**: this will create an infinite loop!
* Side effects are allowed (and encouraged) in **event handler functions**!
  + There is also a special hook to **register side effects** (useEffect)

# Lecture: How Events Work in React

## **SYNTHETIC** EVENTS

* Like a wrapper around the DOM’s native event object
* Has **same interface** as native event objects, like stopPropagation() and preventDefault()
* Fixes browser inconsistencies, so that events work in the exact **same way in all browsers**
* Most synthetic events bubble (including focus, blur, and change), except for scroll
* Attributes for event handlers in React are named using **camelCase** (onClick instead of onclick or click)
* Default behavior can **not** be prevented by returning false (only by using preventDefault())
* Attach “Capture” if you need to handle during **capture phase** (example: onClickCapture)

# Lecture: Libraries vs. Frameworks & the React Ecosystem

* Analogy: you wanna make boba tea at home.
  + You can get an overpriced all-in-one kit from the sub-par World Market on Rodeo that’s got everything you need
  + Or you can research and collect separate ingredients from various (probably Asian) supermarkets, so you can fully customize and put together some niche shit, yknow? 😎
* **Framework** – the “all-in-one” kit
  + Pros: **ease of mind**. Everything you need to build a complete application is **included** in the framework
  + Cons: no choice to customize. You’re **stuck with the framework’s chosen tools** and conventions (which isn’t always a bad thing)
* **Library** – “separate ingredients”
  + Pros: freedom! You can **choose multiple 3rd-party libraries** to build a complete application
  + Cons: decision fatigue… you need to **research, download, learn,** and **stay up-to-date** with multiple external libraries

## REACT 3RD PARTY LIBRARY ECOSYSTEM

A screenshot of a computer

Description automatically generated

## FRAMEWORKS BUILT **ON TOP** OF REACT

* React frameworks offer many other features:
  + Server-side rendering (SSR), static site generation (SSG), better developer experience (DX), etc.

A diagram of a application

Description automatically generated

## PRACTICAL SUMMARY

* A **component** is like a blueprint for a piece of UI that will eventually exist on the screen. When we “use” a component, React creates a **component instance**, which is like an actual physical manifestation of a component, containing props, state, and more. A component instance, when rendered, will return a React element.
* “Rendering” only means **calling component functions** and calculating what DOM elements need to be inserted, deleted, or updated. It has nothing to do with writing to the DOM. Therefore, **each time a component instance is rendered and re-rendered, the function is called again.**
* Only the **initial app render** and **state updates** can cause a render, which happens for the **entire application**, not just one single component
* When a component instance gets re-rendered, **all its children will get re-rendered as well**. This doesn’t mean that all children will get updated in the DOM, thanks to reconciliation, which checks which elements have actually changed between two renders. But all this re-rendering can still have an impact on performance (more on that later in the course)
* Diffing is how React decides which DOM elements need to be added or modified. If, between renders, a certain React element **stays at the same position in the element tree**, the corresponding DOM element and component state will stay the same. If the element **changed to a different position**, or if it’s a **different element type**, the DOM element and state will be destroyed
* Giving elements a key prop allows React to distinguish between multiple component instances. **When a key stays the same across renders**, the element is kept in the DOM. This is why we need to use keys in lists. **When we change the key between renders**, the DOM element will be destroyed and rebuilt. We use this as a **trick to reset state**
* **Never declare a new component inside another component!** Doing so will re-create the nested component every time the parent component re-renders. React will always see the nested component as **new**, and therefore **reset its state** each time the parent state is updated
* The logic that produces JSX output for a component instance (“render logic”) is **not allowed to produce any side effects**: no API calls, no timers, no object or variable mutations, no state updates. **Side effects are allowed in event handlers** and **useEffect** (more on this in next section)
* The DOM is updated in the commit phase, **but not by React, but by a “renderer” called ReactDOM**. That’s why we always need to include both libraries in a React web app project. We can use other renderers to use React on different platforms, for example to build mobile or native apps
* Multiple state updates inside an event handler function are **batched**, so they happen all at once, **causing only one re-render**. This means we can **not access a state variable immediately after updating it**: state updates are **asynchronous**. Since React 18, batching also happens in timeouts, promises, and native event handlers.
* When using events in event handlers, we get access to a **synthetic event object**, not the browser’s native object, so that **events work the same way across all browsers**. The difference is that **most synthetic events bubble**, including focus, blur, and change, which do not bubble as native browser events. Only the scroll event does not bubble.
* React is a library, not a framework. This means that you can assemble your application using your favorite third-party libraries. The downside is that you need to find and learn all these additional libraries. No problem, as you will learn about the most commonly used libraries in this course

SECTION 12: EFFECTS AND DATA FETCHING

# Lecture: the Component Lifecycle

* We can define code to run at these specific points in time

1. Mount / Initial Render
2. Re-Render (*optional*)
3. Unmount

## Mount / Initial Render

* Component instance is rendered for the first time
* Fresh state and props are created

## Re-Render

* Happens when:
  + **State** changes
  + **Props** change
  + **Parent** re-renders
  + **Context** changes

## Unmount

* Component instance is **destroyed** and **removed**
* State and props are **destroyed**

# Lecture – How NOT to Fetch Data in React

Refer back to 07-use-popcorn, copy App.js and save as App-v1.js to hold on to original scripting

* Here’s how NOT to fetch data using React

# Lecture – A First Look at Effects