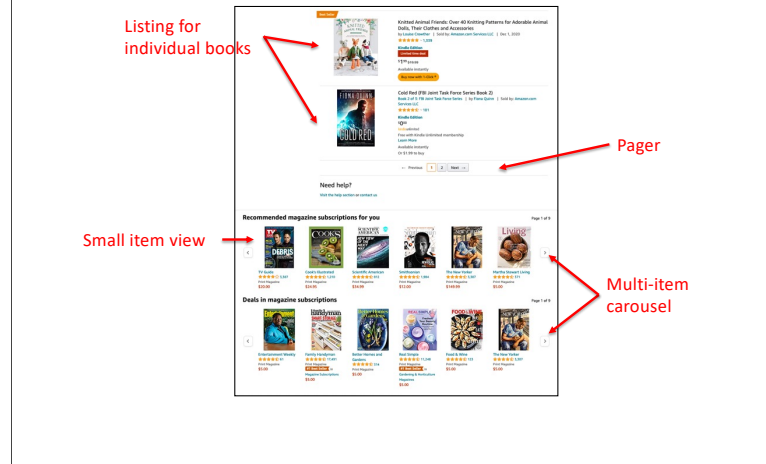Here is an example from Amazon. If we opened the developer tools to view the page, it would look like one large "chunk" of HTML, but that is not how the page is designed or implemented. Instead, it is implemented as a set of components that can be composed together to create the final page (application). There are all kinds of components on this page: individual book listings, the pager, the multi-item carousels, the small item views. Each one of these entities is a component. They all have some abstract form, some associated data that changes their appearance, and potentially some interactive functionality.

This is largely how modern web development is done, by composing "smart" modular components rather than hand coding everything in raw HTML.

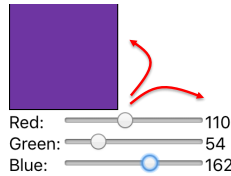"Color picker" component example

Red: 110
Green: 54
Blue: 162
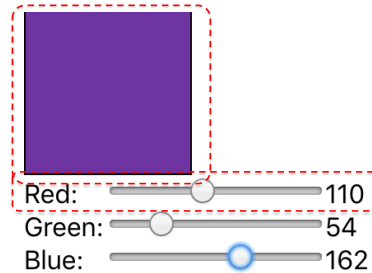
Consider a Color Picker… This is a component that we might design for a page. It has a swatch that shows the current color and three sliders for changing the RGB values. What are some possible sub-components? The color swatch and the slider (for each color component).

As we start to think about our applications in terms of components, it is not just thinking about how we might "split" up the way the page is displayed, it also thinking about data, or specifically state. What information do our components maintain and how might it change as the user interacts with the application.

Consider the red channel of the color picker. What data do we have? The color value. How many views of that data do we have? Three: 1) the position of the slider itself, 2) the numeric value label, and 3) the color swatch. All three need to be updated when we change the red value.

Implementing that kind of interactivity is the role for Javascript in the browswer.

Use JS to create interactivity

```html
<div class="blue-slider">
  <div class="color-label">blue: </div>
  <input type="range" id="slider-b" …/>
  <span id="value-b"></span>
</div>
```

Red: 110
Green: 54
Blue: 162

```js
// Set oninput callback for each slider
sliders.forEach((slider) =>
  slider.addEventListener("input", update));

const update = function() {
  colorBox.style.background =
    `rgb(${sliders[0].value}, ${sliders[1].value}, ${sliders[2].value})`;
  sliders.forEach((slider, index) =>
    labels[index].innerHTML = slider.value);
};
```
Update swatch and numeric text

Here is the JS implementation for our color picker., and specifically the key function that updates the color swatch, and numeric value each time a slider is moved, that is keeps everything in sync.

Interactive demo: https://codepen.io/mlinderm/pen/NWqPbeq?editors=1010

Note, I will use the term the DOM frequently. At a high-level the "DOM" is what the browser renders and I will use as a catch-all term for what is shown on the screen. More specifically, a web page (or document) is a set of (many) nested boxes, i.e., nested elements. The Document Object Model (DOM) is the tree data structure representing the nested structure of the page. The boxes (HTML tags in our context) are nodes in the tree. The DOM properties and methods (the API) provide programmatic access to the tree to access or change the document's structure, style or content.

Recall that the browser is *asynchronous*
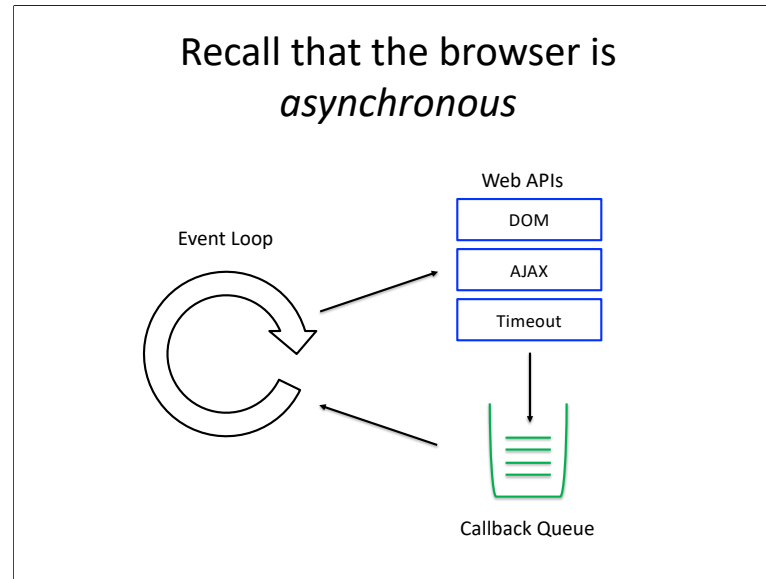
That is when our JavaScript code ran the first time it didn't actually change the color. Instead, it registered a callback to invoked when the user move the slider (the "input" event). Whenever the user moves the slider, that call back, the update function, is invoked. The update function accesses the current value of the slider and uses that value to update the swatch and numeric text.

How to extend the color picker?

RGB Calculator

rgb(255, 25, 0)

#ff1900

hsl(6, 100%, 50%)

Input &
Output

R: 255

255

G: 25

25

B: 0

0

That got a lot more complicated! How do we keep these inputs and outputs in sync with each other (a change to any input should update all outputs)? If we continued with the "vanilla JS" approach, each time we add a new input or output we would need to update the callbacks to update all other inputs/outputs. Simple enough so far, but that quickly becomes tricky.

Different "design patterns" for the same problem

BACKBONE.JS

Vue.js

A NGULAR

React

- Event based (e.g., Backbone)
  - Changing the data triggers an event
  - Views register event handlers
- Two-way binding (e.g., Angular)
  - Assigning to a value propagates to dependent components and vice versa
- Efficient re-rendering (e.g., React)
  - Re-render all subcomponents when data changes

The challenge of keeping everything in sync is not a new problem… Some solutions that have been/are in use.

Philosophy of React

1. Render the UI as it should appear for any given state of the application
2. Update the state in response to user actions
3. Repeat (i.e., re-render UI with new state)

The key conceptual idea is that those two steps are now decoupled and so simpler

The key technical enabler was efficient re-rendering when the data changes

React is a framework (library) designed to help us solve this exact problem. That is build highly interactive and "reactive" UIs. The key idea is to decouple the render of the current state from the updates to that state. You just need to answer those different questions and do so separately. What do I want to the UI to look like at any given moment – more formally for any given state of the application – and how do I update that state based on user actions. We don't have to answer the much trickier question of how do we want to update the UI in response to user actions. React takes care of efficiently propagating those state changes to (throughout) the UI.

What is the state in the "extended" color picker?

A. The current color components

B. A *and* the slider positions

C. B *and* the numeric text inputs

D. C *and* the outputs, e.g., hex output

Answer: A

By state we mean what information/data do I need to uniquely specify the UI. In this case there is only one piece of information needed to uniquely specify the UI – the RGB color components. All the inputs and outputs are determined by that information (that is the positions of the sliders, the swatch color, etc.) and should all show the same information.

What is the state in our "enhanced" color picker?

State in React is the answer to the question:
What information do I need to uniquely specify the UI?

```
const [red, setRed] = useState(0);
const [green, setGreen] = useState(0);
const [blue, setBlue] = useState(0);
```

That's it! Even in the most complex color picker, the only state is the 3 current color components. Every aspect of the UI depends on those three values. Here we implement that state using Hooks. At its simplest, we can create state with the useState() function. This returns an array with a constant value (the current value for our state object, initialized to the value we pass into useState()) and a setter function for updating the state (e.g., setRed). We can't and shouldn't change the value, instead we use the setter function. When we call that setter function it will cause the component to re-render to update the UI based on the new state value.

What is an implication of that re-render? Specifically, what would happen if we called the state update in the function that renders our component? An infinite loop!

JS note: This is an example de-structuring assignments, that is assigning the elements in the array returned by useState to individual variables.

1. Render the UI for a given state

```
const [red, setRed] = useState(0);
…
```

Red: ○━━━━━━━━ 0 ⌄
Green: ○━━━━━━━━ 0 ⌄
Blue: ○━━━━━━━━ 0 ⌄
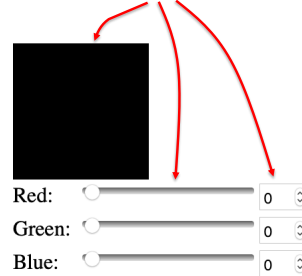
Let's revisit the steps of the philosophy of React in the context of the color picker.
Step1 : The state – the 3 color components – determines the color of the color
swatch, the position of the slider and the value in the numeric output. We first make
sure we can render those values in our UI.

2. Update the state, then re-render

```
const [red, setRed] = useState(0);
…
```

Red: ⬡──────────────   0 ⬍
Green: ⬡──────────────   0 ⬍
Blue: ⬡──────────────   0 ⬍
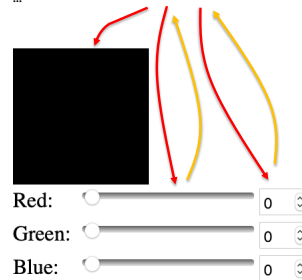
Step 2: We will then connect these components such that changing the slider bar changes the state (e.g., via the setRed setter) – i.e., the orange arrows.

Step 3: Setting the state will trigger React to re-render, but with a new value of state. That new value of state will propagate (via the red arrows) to create the new view (with the updated color value).

Is the slider aware of the numeric input (or the swatch)? No. Each component only needs to know the state and how to update that state where relevant! That is how React helps us minimize the complexity when building complex UIs. The complexity in effect grows linearly with the interactions we add, instead of exponentially with the interactions between those interactions.

"Thinking in React"

1. Break the UI into a component hierarchy
2. Build a static version in React
3. Identify the minimal (but complete) representation of state
4. Identify where your state should live
5. Add "inverse" data flow (data flows down, callbacks flow up)

https://react.dev/learn/thinking-in-react

Break the UI into components

ColorPicker

LabeledSlider

Red:	⬤—————	0 ⬍
Green:	⬤—————	0 ⬍
Blue:	⬤—————	0 ⬍
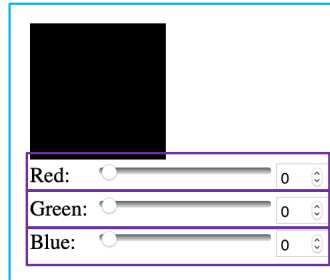
*A component is function that takes a single argument, the **props**, and returns a hierarchy of components*

The fundamental unit of React is the component. In React, we can implement components as either *classes* or *functions*. For our purposes, we will primarily stick to function-based components, but many older examples you find online will use classes.

A function-based component is a function that takes a single argument, termed the **props**, and returns a hierarchy of components (think of these child components like a nested tree, similar to the DOM itself) with a single root. The root element returned by the function is what is added to the virtual DOM.

The first step in building a React app is break down the UI (the view) into a hierarchy of components and sub-components. In the color picker there is one main component (the color picker itself, with the swatch) and the 3 sliders and corresponding value display/inputs.

Explore this starting point at: https://codepen.io/mlinderm/pen/YzXwNdd

Concisely expressing the view: JSX

```
// Example HTML
var heading = React.createElement("h1", null, "Hello, world!");
// Example component
var person = React.createElement(Person, {
  name: p.name,                              Props
  address: p.addr
});
```
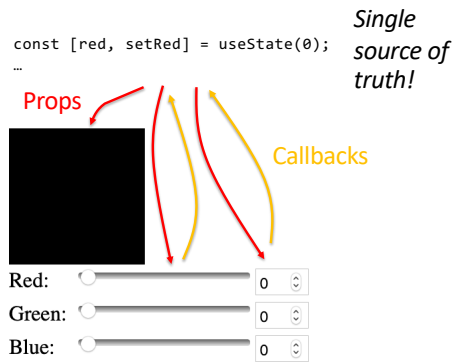
```
// Example HTML
const heading = <h1>Hello, world!</h1>;
// Example component            {embedded JS}
const person = <Person name={p.name} address={p.addr} />;
```

Props

JSX is an extension to Javascript for efficiently describing the UI, including both React components (like Person) and HTML (like h1). Since JSX is an extension to JavaScript, we will need a compiler to convert it to standard JavaScript. The online sandboxes do that for us (as an option) and the tool we will use for setting up React application (e.g., Next) integrates the necessary compiler to *transpile* JSX (and support features of ES6). We will use JSX in our components (as it is much more concise and clearer). However, you should realize that it is being translated directly into normal JavaScript functions (i.e., it is "syntactic sugar").

Note that the names for our props, "name" and "address" in this example are our choice (excluding some reserved names). What is inside the curly brackets is Javascript, and typically references variables defined in our component function.

Props flow down, callbacks flow up

```
const [red, setRed] = useState(0);
…
```

Single source of truth!

Props

Callbacks

Red: [slider] [0]
Green: [slider] [0]
Blue: [slider] [0]

Recall that our state is just the 3 color components. Where should this state live (step 4 in "Thinking in React")? We need this information in the sliders, i.e., in LabeledSlider, but also in ColorPicker to set the swatch color. Per the React documentation: "Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor." Thus, we will implement the state in the ColorPicker component (the closest common ancestor).

That state then "flows down" to the labeled sliders as props to those components. React components must act like pure functions with respect to their props. That is a component can't modify its props (this enables efficient updates). To communicate updates "back up" we supply a callback to the child that modifies the state in the parent (the "inverse" data flow or step 5 in Thinking in React).

Some important notes about modifying state:
• Do not modify state directly, instead use the setter.
• State updates may be asynchronous. React may batch updates, and so you shouldn't assume the state has immediately changed after the call to the setter.

Putting it all together: the ColorPicker

```
function ColorPicker() {
  const [red, setRed] = useState(0);
  const [green, setGreen] = useState(0);
  const [blue, setBlue] = useState(0);

  const color = {
    background: `rgb(${red}, ${green}, ${blue})`
  };
  return (
    <div>
      <div className="color-swatch" style={color} />
      <LabeledSlider label="Red" value={red} setValue={setRed} />
      <LabeledSlider label="Green" value={green} setValue={setGreen} />
      <LabeledSlider label="Blue" value={blue} setValue={setBlue} />
    </div>
  );
}
```

Template literal

Passing state as prop

Passing a Callback as prop

Check out a demo of the complete implementation at:
https://codepen.io/mlinderm/pen/JjdYmOK

JS note: This includes an example of a template literal where we dynamically construct a string from JS variables.

16

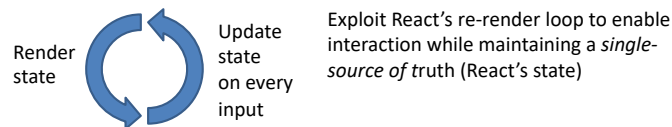## Forms (inputs) are implemented as "controlled components"

Destructure props object argument into variables

```
function LabeledSlider({ label, value, setValue}) {
 return (
  <div>
   <span>{label}</span>
   <input type="range" min="0" max="255"
    value={value}
    onChange={event => setValue(parseInt(event.target.value, 10))}
   />
   <span>{value}</span>
  </div>
 );
}
```

Input value determined by React state (or props derived from state)

Any change updates state, which re-renders input with new value

Render state Update state on every input

Exploit React's re-render loop to enable interaction while maintaining a *single-source of t*ruth (React's state)

As you review the code, note that we use "controlled" <input> components. Controlled components are form elements with state controlled by React. Uncontrolled components maintain their own state. The latter is the way <input> elements naturally work (recall the "vanilla JS" color picker). The former, "controlled", is the recommended approach as it ensures there is only one source of truth, the React state. We set the <input> element's value from state and provide an onChange (or other relevant) handler to update that state in response to user input. Each state change triggers a re-rendering that shows the changes the user just initiated.

JS note: This includes an example of destructing the single props object argument into individual variables.
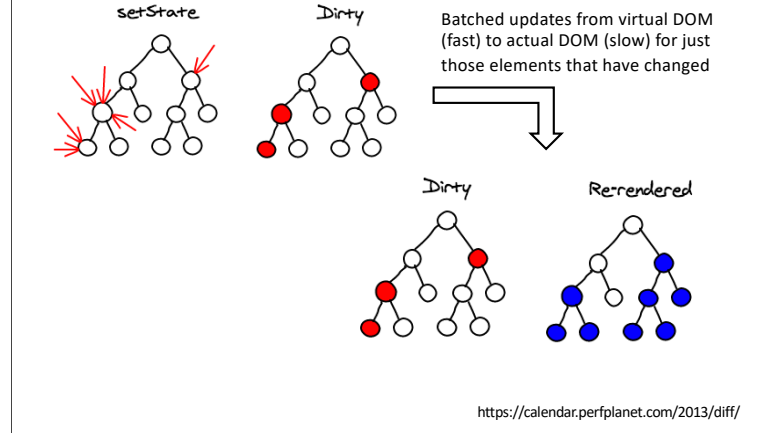
A key innovation in React is making that re-rendering process very fast. React maintains a virtual DOM that represents the ideal state of the UI. Changing the application state triggers re-rendering, which changes the virtual DOM (those changes are fast since only the "virtual" DOM is changing). Any differences between the virtual DOM and actual DOM are then reconciled to the bring the actual DOM to the desired state. But only those elements that changed are updated making this process more efficient.

Further, state updates may be asynchronous. React may batch updates. Thus, you shouldn't assume the state has immediate changed after the call to the state setter.

Often re-rendering is sufficiently fast that we don't need to worry about when and how components are re-rendering. And that should be the starting point. If, however, we observe performance problems (and only if we observe problems) we can optimize the rendering behavior (avoid unnecessary re-renders). See the "read more" links for additional information about how re-rendering works and how we could optimize when components re-render.

Why React: Design patterns

The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander

Even our simple color picker starting getting complex. As we tackle more sophisticated applications, we will clearly need approaches to manage/mitigate SW complexity. One approach is *design patterns*.

Effectively, a design pattern describes those aspects of a problem and solution that are the same every time (and thus can be DRY'd up!*). A design pattern is not a particular class or library, it is a template. You will build up a "library" of these templates over time. React is an implementation of design pattern for building interactive UIs. The operations on the (virtual) DOM are the "part that is the same" and occur entirely "behind the scenes" within React. As a developer your focus is just on rendering the desired UI. That is, you can focus on the part that is different each time instead of the parts that are the same.

* DRY is an acronym for ”Don't Repeat Yourself”, i.e., don't duplicate code/work

Design pattern: Model-View-Controller
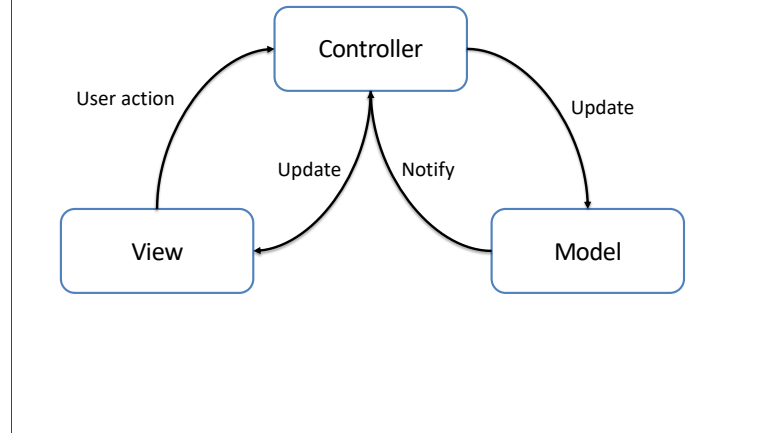
In an even more general sense, React is trying to solve the problem of what do we show the user (what should appear on the screen) in a graphical application and how does that "view" change in response to user actions. A more general design pattern for that problem is the Model-View-Controller pattern (widely used in web applications, but also GUIs). MVC separates the data/resource (Model) from the presentation (View) with the Controller. Generally, the controller manipulates the model in response to user actions and presents the resulting model(s) for rendering by the view(s). I say generally because there are many different implementations of MVC, all of which have slightly different MVC roles. There are also other related patterns like MVVM – Model View ViewModel that divide up responsibilities slightly differently.

Where does React fit into this pattern? At a high-level, it is just the "V" (the view) (although not all would agree with that characterization), with the server (something we will talk about in subsequent classes) responsible for the C and M. The reality is a little less clear cut. Our React components will have elements of V and C (we have already seen that...). As with other design patterns, the value is in the "core" of the solution, that is thinking about how to separate/decoupling the roles of storing the state of the application (the model), how we visualize that state (the view) and how we modify the state of the application in response to user actions (the controller). If we think about those roles explicitly, we are more likely to produce "beautiful" code.

In some frameworks, that decomposition is not so "optional", that is the framework is really built around this design pattern, and so we need to explicitly identify those components of our application.

Symptoms of anti-patterns, i.e., tactical programming or a sign it's going awry

- Viscosity
   Easier to do a hack than do the "Right Thing"
- Immobility
   Can't DRY out functionality
- Needless repetition
- Needless repetition
- Needless complexity from generality

There are also anti-patterns, that is code that looks like it should probably follow some design pattern but doesn't. Such code is both the cause and result of "technical debt". Some symptoms of anti-patterns…

These are more specific manifestations of the tactical programming we discussed previously and signs that complexity is winning: Change is hard, high-cognitive local and unknown unknowns.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license

Design patterns (or abstractions): Moderation in all things

Two key questions:

1. What is benefit of this pattern (abstraction)?
2. What is the cost of this pattern (abstraction)?

If we can't answer those questions we don't know if we are coming out ahead. What if we are paying a cost, but not getting any benefit, that is paying for a solution to a problem we don't have.

Using a particular design pattern or abstraction (we can think of React abstracting away the details of updating our UI in response to state changes) doesn't automatically make our software better, it only does so if it fits our problem and solves a problem we actually have. Our course webpage doesn't use React. Why? Because it is almost all static (no state to interactively update!)

How do we know if our problem is a good fit? As a start, we want to keep an eye out for the anti-patterns we just discussed. A positive sign is that we are successfully abstracting away unimportant details. The word unimportant is crucial (and sometimes hard to define). In this context, we would describe the mechanics of efficiently updating the DOM as unimportant. Another sign is that a good abstraction will have a simple interface, but the functionality behind that interface is "deep". The interface for managing state with React is very simple, but the functionality behind the rendering process is substantial ("deep").

https://kentcdodds.com/blog/how-to-react
Ousterhout, John K. . A Philosophy of Software Design, 2nd Edition