I will use the term the DOM a lot. 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.
That is when our javascript code ran the first time it didn’t do much in terms of actually setting the color. Instead it registered a callback to invoked when the user move the slider (the “oninput” 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.
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.
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 to different questions. 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 the the UI.

Philosophy of React

1. Render the UI as it should appear for any given state of the application
2. Update the state as a result of user actions
3. Repeat...

*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*
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 of the inputs and outputs are tied to that piece of information (that is the slider, the swatch, etc.) and should all show the same information the current color components.
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?*

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

That’s it! Even if 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 (a more recent addition to React). 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 will return to this implementation detail shortly.
1. Render the UI for a given state

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

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.
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. 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).

What do you notice? 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!
“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://reactjs.org/docs/thinking-in-react.html
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 examples you find online will use classes (and we will look at some class examples ourselves).

A function-based component is a function that takes a single argument, which we refer to as the props, and returns a hierarchy of components (think of these children 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

```js
// Example HTML
var heading = React.createElement("h1", null, "Hello, world!");

// Example component
var person = React.createElement(Person, {
    name: p.name,
    address: p.addr
});

// Example HTML
const heading = <h1>Hello, world!</h1>;

// Example component
const person = <Person name={p.name} address={p.addr} />;
```

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 (Create React App or CRA) integrates the Babel compiler to transpile JSX (and support features of ES6). We will use JSX in our components (as it is much more concise and clear). However, you should realize that it is being translated into normal JavaScript functions.
Recall that our state is just the three 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 actually changed after the call to the setter.
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.
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 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 actually changed after the call to the setter.
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.
Different “design patterns” for the same problem

- 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
A well-known example of design patterns comes from this very influential book on writing object-oriented software. The book describes a variety of techniques for dealing with common issues that come up in object-oriented design. An example would be the factory method, where you create a static method for creating and initializing new objects instead of a constructor. This allows you to swap in different subclasses depending on need.
The three-tiered architecture is an architectural design pattern, and the way most web apps are designed.
The Model-View-Controller pattern is used for interactive graphical applications like web applications.

Note the similar structure to the three-tier architecture...

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) – with the server 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...).
Which of the following is a benefit of the Model-View-Controller (MVC) design pattern?

A. MVC supports different views for each user/consumer of a model
B. Ensures there's a one-to-one mapping from each model to a single view
C. Provides a window into the model and controller for debugging purposes
D. Like most design patterns, it results in more concise code

Answer: A

MVC separates the representation of data from how it is presented to the user. Many different users can use the application and they each can see different views based on their requests. There can be multiple views associated with a model, MVC doesn't provide any specific introspection of the model or controller, and MVC does not necessarily reduce the amount of code. Design patterns are used to reduce complexity, separate concerns, improve modifiability, improve security, improve performance, etc. but they may or may not result in more concise code.

Adapted from Kristen Walcott-Justice
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...

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