REACT

list
HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form
This form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called controlled components.
Controlled Components
In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with setState().

We can combine the two by making the React state be the single source of truth. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a controlled component.

For example, if we want to make the previous example log the name when it is submitted, we can write the form as a controlled component.
Alternatives to Controlled Components
It can sometimes be tedious to use controlled components, because you need to write an event handler for every way your data can change and pipe all of the input state through a React component. This can become particularly annoying when you are converting a preexisting codebase to React, or integrating a React application with a non-React library. In these situations, you might want to check out uncontrolled components, an alternative technique for implementing input forms.
------***---***---***---***---***---***
key
React keys are useful when working with dynamically created components or when your lists are altered by users. Setting the key value will keep your components uniquely identified after the change.

Using Keys

Let's dynamically create Content elements with unique index (i). The map function will create three elements from our data array. Since key value needs element.
First, let's review how you transform lists in JavaScript.

Given the code below, we use the map() function to take an array of numbers and double their values. We assign the new array returned by map()to the variable doubled and log it.
Rendering Multiple Components
You can build collections of elements and include them in JSX using curly braces {}.

Below, we loop through the numbers array using the Javascript map() function. We return an <li> element for each item. Finally, we assign the resulting array of elements to listItems:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

DOM:

```
ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

Try it on CodePen.

This code displays a bullet list of numbers between 1 and 5.

Basic List Component
Usually you would render lists inside a component.

We can refactor the previous example into a component that accepts an array of numbers and outputs an unordered list of elements.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```
When you run this code, youll be given a warning that a key should be provided for list items. A key is a special string attribute you need to include section.

Let's assign a key to our list items inside numbers.map() and fix the missing key issue.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```
______***___***___***___***___***___***

lifecycle

Lifecycle Methods

componentWillMount is executed before rendering, on both server and client side.

componentDidMount is executed after first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution like setTimeout or setInterval. We are using it to update the state so we can trigger the other lifecycle methods.

componentWillReceiveProps is invoked as soon as the props are updated before state.

shouldComponentUpdate should return true or false value. This will determine if component will be updated or not. This is set to true by default. If you are sure that value.

componentWillUpdate is called just before rendering.

componentDidUpdate is called just after rendering.

componentWillUnmount is called after the component is unmounted from the dom. We are unmounting our component in main.js.

In our example we are setting initial state in constructor function. The setNewnumber is used to update the state. All the lifecycle methods are inside Content component.
______***___***___***___***___***___***

ref

React supports a special attribute that you can attach to any component. The ref attribute takes a callback function, and the callback will be executed immediately after the component is mounted or unmounted.
When the ref attribute is used on an HTML element, the ref callback receives the underlying DOM element as its argument.

In the typical React dataflow, props are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch
There are a few good use cases for refs:
•”Ö æ v–ær fö7W2Â FPxt selection, or media playback.
••@riggering imperative animations.
•”–çFPgrating with third-party DOM libraries.

React will call the ref callback with the DOM element when the component mounts, and call it with null when it unmounts.
Using the ref callback just to set a property on the class is a common pattern for accessing DOM elements.

•F†R &VfW'&VB pay is to set the property in the ref callback like in the above examp
There is even a shorter way to write it: ref={input => this.textInput = input}.
•
•v†Vâ F†R &Vb GG&– ute is used on a custom component declared as a class, the refcallback receives the mounted instance of the component as its argument.
______***___***___***___***___***___***

composition

Composition vs Inheritance
React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

In this section, we will consider a few problems where developers new to React often reach for inheritance, and show how we can solve them with composition.

Containment
Some components don't know their children ahead of time. This is especially common for components like Sidebar or Dialog that represent generic boxes.

We recommend that such components use the special children prop to pass children elements directly into their output
Anything inside the <FancyBorder> JSX tag gets passed into the FancyBorder component as a children prop. Since FancyBorder renders {props.children} inside a <div>, the passed elements appear in the final output.

While this is less common, sometimes you might need multiple holes in a component. In such cases you may come up with your own convention instead of using children
Sometimes we think about components as being special cases of other components. For example, we might say that a WelcomeDialog is a special case of Dialog.

In React, this is also achieved by composition, where a more specific component renders a more generic one and configures it with props
______***___***___***___***___***___***
render
Rendering Elements
Elements are the smallest building blocks of React apps.

An element describes what you want to see on the screen:

const element = <h1>Hello, world</h1>;
Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.

Note:
One might confuse elements with a more widely known concept of components. We will introduce components in the next section. Elements are what components are made of, and we encourage you to read this section before jumping ahead.
Rendering an Element into the DOM.
Applications built with just React usually have a single root DOM node. If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.

To render a React element into a root DOM node, pass both to ReactDOM.render():

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Try it on CodePen.

Updating the Rendered Element

React elements are immutable. Once you create an element, you can't change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

With our knowledge so far, the only way to update the UI is to create a new element, and pass it to ReactDOM.render().

Consider this ticking clock example:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Try it on CodePen.

It calls ReactDOM.render() every second from a setInterval() callback.

Note:
In practice, most React apps only call ReactDOM.render() once. In the next sections we will learn how such code gets encapsulated into stateful components.
We recommend that you don't skip topics because they build on each other.

React Only Updates What's Necessary

React DOM compares the element and its children to the previous one, and only

applies the DOM updates necessary to bring the DOM to the desired state.
______***___***___***___***___***___***

state

There are two types of data that control a component: props and state. props are set by the parent and they are fixed throughout the lifetime of a component. For data that is going to change, we have to use state.

In general, you should initialize state in the constructor, and then call setState when you want to change it.

For example, let's say we want to make text that blinks all the time. The text itself gets set once when the blinking component gets created, so the text itself is a prop. The state.

In a real application, you probably won't be setting state with a timer. You might set state when you have new data arrive from the server, or from user input. You can also use a state container like Redux to control your data flow. In that case you would use Redux to modify your state rather than calling setState directly.

When setState is called, BlinkApp will re-render its Component. By calling setState within the Timer, the component will re-render every time the Timer ticks.

State works the same way as it does in React, so for more details on handling state, you can look at the React.Component API. At this point, you might be annoyed that most of our examples so far use boring default black text.
______***___***___***___***___***___***

component

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called props) and return React elements describing what should appear on the screen.
Functional and Class Components
The simplest way to define a component is to write a JavaScript function:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

This function is a valid React component because it accepts a single props object argument with data and returns a React element. We call such components 'functional' because they are literally JavaScript functions.

You can also use an ES6 class to define a component:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```
The above two components are equivalent from React's point of view.

Classes have some additional features that we will discuss in the next sections. Until then, we will use functional components for their conciseness.

Rendering a Component
Previously, we only encountered React elements that represent DOM tags:

```
const element = <div />;
```
However, elements can also represent user-defined components:

```
const element = <Welcome name='Sara' />;
```
When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object 'props'.

For example, this code renders 'Hello, Sara' on the page:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name='Sara' />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```
Try it on CodePen.

Let's recap what happens in this example:

We call ReactDOM.render() with the <Welcome name='Sara' /> element.
React calls the Welcome component with {name: 'Sara'} as the props.
Our Welcome component returns a <h1>Hello, Sara</h1> element as the result.
React DOM efficiently updates the DOM to match <h1>Hello, Sara</h1>.
Caveat:
Always start component names with a capital letter.
For example, <div /> represents a DOM tag, but <Welcome /> represents a component and requires Welcome to be in scope.
Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.

For example, we can create an App component that renders Welcome many times:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name='Sara' />
      <Welcome name='Cahal' />
      <Welcome name='Edite' />
    </div>
  );
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Try it on CodePen.

Typically, new React apps have a single App component at the very top. However, if you integrate React into an existing app, you might start bottom-up with a small component like Button and gradually work your way to the top of the view hierarchy.

______***___***___***___***___***___***