1. The render method returns a description of what you want to see on the screen. React takes the description and displays the result.
2. Most React developers use a special syntax called “JSX” which makes these structures easier to write.
3. JSX comes with the full power of JavaScript. You can put any JavaScript expressions within braces inside JSX. Each React element is a JavaScript object that you can store in a variable or pass around in your program.
4. To “remember” things, components use **state**.
5. this.state should be considered as private to a React component that it’s defined in.
6. In [JavaScript classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes), you need to always call super when defining the constructor of a subclass. All React component classes that have a constructor should start with a super(props) call.
7. All React component classes that have a constructor should start with a super(props) call.
8. When you call setState in a component, React automatically updates the child components inside of it too.
9. **The parent component can pass the state back down to the children by using props; this keeps the child components in sync with each other and with the parent component.**

Introducing JSX

* React [doesn’t require](https://reactjs.org/docs/react-without-jsx.html) using JSX, but most people find it helpful.
* Don’t put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.
* const element = <div tabIndex="0"></div>;
* const element = <img src = {user.img}></img>;
* If a tag is empty, you may close it immediately with />, like XML:
  + const element = <img src = {user.img} />;
* Children in jsx
  + const element = (
  + <div>
  + <h1> Heading1 </h1>
  + <h2> Heading2 </h2>
  + </div>
  + );
* JSX prevents injection attacks

Rendering Elements

* An element describes what you want to see on the screen.
* 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.
* React elements are [immutable](https://en.wikipedia.org/wiki/Immutable_object). 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()](https://reactjs.org/docs/react-dom.html" \l "render).
* Thinking about how the UI should look at any given moment, rather than how to change it over time, eliminates a whole class of bugs.
* 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.

Components and Props

* 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.
* When React sees an element representing a user-defined component, it passes JSX attributes and children to this component as a single object. We call this object “props”.
  + function Welcome(props){
  + return (<h1> Welcome {props.username} </h1>);
  + }
  + const tag = <Welcome username="Vatsal" />;
  + ReactDOM.render(
  + tag,
  + document.getElementById('root')
  + );
* Always start component names with a capital letter.
* Typically, new React apps have a single App component at the very top.
* A good rule of thumb is that if a part of your UI is used several times (Button, Panel, Avatar), or is complex enough on its own (App, FeedStory, Comment), it is a good candidate to be extracted to a separate component.
* Whether you declare a component [as a function or a class](https://reactjs.org/docs/components-and-props.html#function-and-class-components), it must never modify its own props.
* functions are called [“pure”](https://en.wikipedia.org/wiki/Pure_function) because they do not attempt to change their inputs.
* All React components must act like pure functions with respect to their props.
* State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.

States and Lifecycle

* State is similar to props, but it is private and fully controlled by the component.
* We will move the date from props to state in three steps:
  + Replace this.props.date with this.state.date in the render() method.
  + Add a [class constructor](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes#Constructor) that assigns the initial this.state.
  + Remove the date prop from the <Clock /> element while rendering.
* The componentDidMount() method runs after the component output has been rendered to the DOM. This is a good place to set up a timer.
* While this.props is set up by React itself and this.state has a special meaning, you are free to add additional fields to the class manually if you need to store something that doesn’t participate in the data flow (like a timer ID).
* The setState() call, allows react to know the state has changed, and calls the render() method again to learn what should be on the screen.
* There are three things you should know about setState():
  + Do not modify state directly. Instead use this.setState({comment: ‘Hello’})
  + State updates may be asynchronous.
  + State updates are merged. The merging is shallow, so this.setState({comments}) leaves this.state.posts intact, but completely replaces this.state.comments.
* Any state is always owned by some specific component, and any data or UI derived from that state can only affect components “below” them in the tree. This is commonly called a “top-down” or “unidirectional” data flow.
* Converting a function to a class:
  + Create an [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes), with the same name, that extends React.Component.
  + Add a single empty method to it called render().
  + Move the body of the function into the render() method.
  + Replace props with this.props in the render() body.
  + Delete the remaining empty function declaration.

Handling Events

* With JSX you pass a function as the event handler, rather than a string.
  + <button onClick = {activateLaser}> Activate Laser </button>
* e.preventDefault() is used to prevent default behavior in React.
* This binding is necessary to make ‘this’ work in the callback.
  + this.handleClick = this.handleClick.bind(this)
* In JavaScript, class methods are not [bound](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind) by default. If you forget to bind this.handleClick and pass it to onClick, this will be undefined when the function is actually called.
* This is not React-specific behavior; it is a part of [how functions work in JavaScript](https://www.smashingmagazine.com/2014/01/understanding-javascript-function-prototype-bind/). Generally, if you refer to a method without () after it, such as onClick={this.handleClick}, you should bind that method.

Conditional Rendering

* Element Variables
  + button = <LoginButton onClick={this.handleClick} />;
  + return ( <div> {button} </div> );
* Inline If with Logical && Operator
  + {unread.length > 0 && <h2> You have {unread.length} messages </h2> }
* Inline If-Else with Conditional Operator
  + <h2> The user is {isLoggedIn ? ‘currently’ : ‘not’} logged in </h2>
  + {isLoggedIn ? <LogoutButton onClick={this.handleClick} /> : <LoginButton onClick={this.handleClick} />}
* Also remember that whenever conditions become too complex, it might be a good time to [extract a component](https://reactjs.org/docs/components-and-props.html#extracting-components).

Lists & Keys

* You can build collections of elements and [include them in JSX](https://reactjs.org/docs/introducing-jsx.html#embedding-expressions-in-jsx) using curly braces {}.
  + const numbers = [1,2,3,4,5];
  + const listItems = numbers.map((number) =>
    - <li> { number } </li>
  + );
  + ReactDOM.render(
    - <ul> {listItems} </ul>,
    - document.getElementById(‘root’)
  + );
* A “key” is a special string attribute you need to include when creating lists of elements.
* Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity.
  + <li key={number.toString()}> { number } </li>
  + <li key={todo.id}> { todo } </li>
  + <li key={index }> { todo } </li> //Not recommended if order of elements may change
* If you choose not to assign an explicit key to list items then React will default to using indexes as keys.
* A good rule of thumb is that elements inside the map() call need keys.
* Keys used within arrays should be unique among their siblings. However they don’t need to be globally unique. We can use the same keys when we produce two different arrays.
* Keys serve as a hint to React but they don’t get passed to your components. If you need the same value in your component, pass it explicitly as a prop with a different name:
  + const content = posts.map((post) =>
    - <Post
      * key = {post.id}
      * id = {post.id}
      * title = {post.title} />
  + );
* With the example above, the Post component can read props.id, but not props.key.

Forms

* While this means you have to type a bit more code, you can now pass the value to other UI elements too, or reset it from other event handlers.
* Since setState() automatically [merges a partial state into the current state](https://reactjs.org/docs/state-and-lifecycle.html#state-updates-are-merged), we only needed to call it with the changed parts.
* <https://reactjs.org/docs/forms.html> - Important to read
* **Study about Formik**

Lifting State Up

* There should be a single “source of truth” for any data that changes in a React application.
* Usually, the state is first added to the component that needs it for rendering.
* Then, if other components also need it, you can lift it up to their closest common ancestor.
* Instead of trying to sync the state between different components, you should rely on the [top-down data flow](https://reactjs.org/docs/state-and-lifecycle.html#the-data-flows-down).
* It takes less work to find and isolate bugs.
* Additionally, you can implement any custom logic to reject or transform user input.
* If something can be derived from either props or state, it probably shouldn’t be in the state.

Composition and Inheritance

* React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.
* Containment
  + Some components don’t know their children ahead of time.
  + We recommend that such components use the special children prop to pass children elements directly into their output.
  + This lets other components pass arbitrary children to them by nesting the JSX.
* Specialization
  + 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.
* Props and composition give you all the flexibility you need to customize a component’s look and behavior in an explicit and safe way.

Thinking In React

* Break the UI into components.
  + Use the same techniques for deciding if you should create a new function or object. One such technique is the [single responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle), that is, a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.
* After getting all the components, build a hierarchy using them.
* Build a static version of site in react.
  + The easiest way is to build a version that takes your data model and renders the UI but has no interactivity.
  + Don’t use state at all to build this static version. State is reserved only for interactivity, that is, data that changes over time.
  + In simpler examples, it’s usually easier to go top-down, and on larger projects, it’s easier to go bottom-up and write tests as you build the components.
  + The components will only have render() methods since this is a static version of your app.
* Identifying The Minimal Representation of the UI State
  + To build your app correctly, you first need to think of the minimal set of mutable state that your app needs.
  + The key here is [DRY: Don’t Repeat Yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself).
  + Figure out the absolute minimal representation of the state your application needs and compute everything else you need on-demand.
  + Ask three questions about each piece of data:
    - Is it passed in from a parent via props? If so, it probably isn’t state.
    - Does it remain unchanged over time? If so, it probably isn’t state.
    - Can you compute it based on any other state or props in your component? If so, it isn’t state.
  + Left over list of data needs to be stored in state.
* Identifying where your state should live
  + For each piece of state in your application:
    - Identify every component that renders something based on that state.
    - Find a common owner component (a single component above all the components that need the state in the hierarchy).
    - Either the common owner or another component higher up in the hierarchy should own the state.
    - If you can’t find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common owner component.
* Add Inverse Data Flow