# How to State

If React is **declarative**, how do we manage state?

- **hooks**!
    - Outside functions to read/write state changes
- Renders JSX with current state
- Event listeners (using onXXX) update state
- JSX Automatically rerenders when state changes

# Input Example

```
import { useState } from 'react';

function App() {
  const [name, setName] = useState('');
  return (
    <div className="app">
      <p>Last name seen was {name}</p>
      <label>
        <span>Name: </span>
        <input
          value={name}
          onInput={ (e) => setName(e.target.value) }
        />
      </label>
    </div>
  );
}
export default App;
```

# SO MUCH - import

```
import { useState } from 'react';
```

This is one of those "other" ways to import

- A file can have one "default" export
  - `import` and give a name of your choice
- A file can have many "named" exports
  - that you import inside `{}` using their name
  - you can change it with `as`:

```
import { useState as someOtherVar } from 'react';
```

- importing from a library (react) involves no path

# SO MUCH - array destructure

`const [name, setName] = useState('');`

`useState()` returns an array

Above code is the same as:

```
const returnedArray = useState('');
const name = returnedArray[0];
const setName = returnedArray[1];
```

`useState()` *always* returns two values

- We destructure to declare and assign 2 variables

# SO MUCH - useState returns

`useState()` always returns two values:

- a value
- a setter function

The value is the last value set with setter function

- defaults to value passed to `useState()`
- value passed to `useState()` ignored once setter called

# SO MUCH - automatic rerender

When a state setter function is called

- output re-renders
- no need to call render()
- Component IS a render() function

# SO MUCH - onInput

```
<input
  value={name}
  onInput={ (e) => setName(e.target.value) }
/>
```

- `name` will always be latest value
- `onInput()` runs whenever there is typing
    - Including backspace/delete
- `e.target` is the input field here
- Notice the self-closing input tag!
    - React translates to actual HTML

# Putting the Parts together

- When App() is called (when `<App/>` renders
    - `name` is set to `''`
    - HTML renders to the screen
    - `<input>` has value `''`
- User types 'J'
    - `onInput` callback fires
        - calls `setName` with 'J'
- Change in state triggers rerender (App() is called)
    - `name` is set to 'J'
    - HTML renders `<input>` with value = 'J'

# Why State?

Remember the concept we are using

- State is variable(s) of values that can change
- **Rendering** is setting HTML based on state
- Events will change state
- After state changes, **render**

True both in React and in advanced plain JS SPAs

Every component defines part of HTML

- Based on state and props

# Revisit Example

```
import { useState } from 'react';

function App() {
  const [name, setName] = useState('');
  return (
    <div className="app">
      <p>Last name seen was {name}</p>
      <label>
        <span>Name: </span>
        <input
          value={name}
          onInput={ (e) => setName(e.target.value) }
        />
      </label>
    </div>
  );
}
export default App;
```

# Component is output HTML

- Based on current state/props
- Defines event handlers
- Event Handlers can change state
    - Which would cause new **render**
    - Which would reflect updated state

# More Example

```
function App() {
  const [inProgress, setInProgress] = useState('');
  const [saved, setSaved] = useState('');
  return (
    <div className="app">
      <p>Name in progress is {inProgress}</p>
      <p>Last Saved name was {saved}</p>
      <label>
        <span>Name: </span>
        <input
          value={inProgress}
          onInput={ (e) => setInProgress(e.target.value) }
        />
        <button
          type="button"
          onClick={ () => setSaved(inProgress) }
        >Save</button>
      </label>
    </div>
  );
}
```

# Two useState()s

```
const [inProgress, setInProgress] = useState('');
const [saved, setSaved] = useState('');
```

Each `useState()` will track a separate value

- Order in file in meaningful
- You can't put useState() inside an `if() {}`

# Different State Updates

```
<input
  value={inProgress}
  onInput={ (e) => setInProgress(e.target.value) }
/>

<button
  type="button"
  onClick={ () => setSaved(inProgress) }
>Save</button>
```

- One "as you type"
- One "after you click"

# See the State-Render cycle at work

- We have State variables and props
- The output HTML is based on the variables
- User events change the state
- Output HTML is automatically updated
    - Based on new state

Trigger for render was the change in state

- Not the user event
- User event was the trigger for the change in state

# Components can call other components

```
import Switch from './Switch';

function App() {
  const [isOn, setIsOn] = useState(false);
  return (
    <div className="app">
      <Switch isFlipped={isOn}/>
      <button onClick={ () => setIsOn(!isOn) } >Flip</button>
    </div>
  );
}
```

```
function Switch({ isFlipped }) {
  const switchState = isFlipped ? "switch--on" : "";
  return (
    <div className="switch__container">
      <div className={`switch ${switchState}`}/>
    </div>
  );
}
```

# Component calls other component

```
import Switch from './Switch';

function App() {
  const [isOn, setIsOn] = useState(false);
  return (
    <div className="app">
      <Switch isFlipped={isOn}/>
      <button onClick={ () => setIsOn(!isOn) } >Flip</button>
    </div>
  );
}
```

Both `App.jsx` and `Switch.jsx` are components

- No limits to putting them together

# State became a prop

```
const [isOn, setIsOn] = useState(false);
return (
  <div className="app">
    <Switch isFlipped={isOn}/>
```

- `isOn` state passed to `<Switch` as a prop
- name of prop changed! (`isFlipped`)
    - Does not need to change/stay the same
    - Passing a parameter to a function
    - New variable, can be same or different name
    - Does MATTER! A lot!
        - Some names are better changed
        - Some names are better staying the same

# Component ignorant of source of prop

```
function Switch({ isFlipped }) {
  const switchState = isFlipped ? "switch--on" : "";
  return (
    <div className="switch__container">
      <div className={`switch ${switchState}`}/>
    </div>
  );
}
```

- Doesn't know `isFlipped` was set by state
    - That's good. **decoupled**
- Rerendered when parent rerendered
- Notice template literal `` `` with `switchState`
- Used to embed in string

# Showing a list

```
function App() {
  const [todos, setTodos] = useState([
    'Pounce',
    'Chase Laser Pointer',
    'Nap',
  ]);
  return (
    <div className="app">
      <TodoList list={todos}/>
    </div>
  );
}
```

```
function TodoList({ list }) {
  const items = list.map(
    item => ( <li>{item}</li> )
  );
  return (
    <ul className="todos">
      {items}
    </ul>
  );
}
```

# Check the console for errors and warnings!

- Warning: `'setTodos' is assigned a value but never used no-unused-vars`
- Error: `Warning: Each child in a list should have a unique "key" prop.`

Why does the Error say "Warning"? Grr.

- Warnings don't prevent things from working, but may indicate a problem
  - This is coming from the linting tool, which has a rule about unused variables
- Errors indicate something definitely wrong

# Rendered lists and "key" prop

Rendered lists in React need a "key" prop

- React does comparison logic to decide what to actually change in DOM
    - Delete item 5 out of 10: looks like changed 5 items and deleted last
- key props allow to see what really changed
    - must be unique
    - must stay the same between renders
        - generally bad to use index

# Fixing our key prop

```
function TodoList({ list }) {
  const items = list.map(
    item => ( <li key={item}>{item}</li> )
  );
  return (
    <ul className="todos">
      {items}
    </ul>
  );
}
```

- Unique `key` prop added

# Understanding the List

```
function TodoList({ list }) {
  const items = list.map(
    item => ( <li key={item}>{item}</li> )
  );
  return (
    <ul className="todos">
      {items}
    </ul>
  );
}
```

- map list of items to list of JSX elements
- NO JOIN
- NOT A STRING
- embed list in JSX

# How to show different content sometimes

What if you want to have different options for content

- Example: Login form vs content + Logout?

# A Conditional Example

```
const [isLoggedIn, setIsLoggedIn] = useState(false);
const [username, setUsername] = useState('');
return (
  <div className="app">
  { isLoggedIn
    ? <div>
        Hello {username}
        <button onClick={() => setIsLoggedIn(false)}>Logout</button>
      </div>
    : <form>
        <label>
          <span>Username: </span>
          <input value={username} onInput={(e) => setUsername(e.target.value)}/>
        </label>
        <button type="button" onClick={() => setIsLoggedIn(true)}>Login</button>
      </form>
   }
  </div>
);
```

# A Different Conditional Example

```
const [isLoggedIn, setIsLoggedIn] = useState(false);
const [username, setUsername] = useState('');
const content =
( <div>
  Hello {username}
  <button onClick={() => setIsLoggedIn(false)}>Logout</button>
</div>);
const login =
(<form>
  <label>
    <span>Username: </span>
    <input value={username} onInput={(e) => setUsername(e.target.value)}/>
  </label>
  <button type="button" onClick={() => setIsLoggedIn(true)}>Login</button>
</form>);

return (
  <div className="app">
  { isLoggedIn ? content : login }
  </div>
);
```

# Yet Another Conditional Example

```
const [isLoggedIn, setIsLoggedIn] = useState(false);
const [username, setUsername] = useState('');
let content;
if (isLoggedIn) {
  content = ( <div>
    Hello {username}
    <button onClick={() => setIsLoggedIn(false)}>Logout</button>
  </div>);
} else {
  content = (<form>
    <label>
      <span>Username: </span>
      <input value={username} onInput={(e) => setUsername(e.target.value)}/>
    </label>
    <button type="button" onClick={() => setIsLoggedIn(true)}>Login</button>
  </form>);
}

return (
  <div className="app"> { content } </div>
);
```

# Conditional Rendering

- We know Rendering is based on state
    - Output can be different based on state
- We know events can change state

**Our app can show different "pages" based on state**

- Completely different "pages"
- Or just different parts

# State goes "down"

```
function App() {
  const [todos, setTodos] = useState([
    'Pounce',
    'Chase Laser Pointer',
    'Nap',
  ]);
  return (
    <div className="app">
      <TodoList list={todos}/>
    </div>
  );
}
```

- State is passed "down"
    - to children

# What if a child wants to change state?

Child component has no access to setter!

- Cannot reach "up" to variables in parent
- Parent must pass some function to change
  - Direct setter (Ex: `setName`, etc)
  - OR wrapper of direct setter

```
function Demo() {
  const [name, setName] = useState('');

  const onClick = () => { // "wrapper" around setName
    setName('Jorts');
  }

  return (
    <button onClick={onClick}>Unite</button>
  )
};
```

# A Better Conditional Example

```
import Content from './Content';
import Login from './Login';

function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  const [username, setUsername] = useState('');
  return (
    <div className="app">
      { isLoggedIn
        ? <Content
            username={username}
            setLoggedIn={setLoggedIn}
          />
        : <Login
            username={username}
            setUsername={setUsername}
            setLoggedIn={setLoggedIn}
          />
      }
    </div>
  );
}
```

# The other components

```
function Content({ username, setLoggedIn }) {
  return ( <div>
    Hello {username}
    <button onClick={() =>
      setIsLoggedIn(false)}>Logout</button>
  </div>);
}
```

```
function Login({ username, setUsername, setIsLoggedIn }) {
  return ( <form>
    <label>
      <span>Username: </span>
      <input value={username}
        onInput={(e) => setUsername(e.target.value)}/>
    </label>
    <button type="button"
      onClick={() => setIsLoggedIn(true)}>Login</button>
  </form>);
}
```

# You can be more generic

```
  const onLogin = (username) => {
    setUsername(username);
    setIsLoggedIn(true);
  };
  const onLogout = () => setIsLoggedIn(false);

  return (
    <div className="app">
      { isLoggedIn
        ? <Content
            username={username}
            onLogout={onLogout}
          />
        : <Login
            onLogin={onLogin}
          />
      }
    </div>
  );
}
```

# The more generic parts

```
function Content({ username, onLogout }) {
  return ( <div>
    Hello {username}
    <button onClick={onLogout}>Logout</button>
  </div>);
}
```

```
function Login({ onLogin }) {
  const [username, setUsername] = useState('');
  return ( <form>
    <label>
      <span>Username: </span>
      <input value={username}
        onInput={(e) => setUsername(e.target.value)}/>
    </label>
    <button type="button"
      onClick={() => onLogin(username)}>Login</button>
  </form>);
}
```

# Each component can have state

See the `useState()` here!

- Distinct from the username of `App`
- Allows for custom behavior

```
function Login({ onLogin }) {
  const [username, setUsername] = useState('');
  return ( <form>
    <label>
      <span>Username: </span>
      <input value={username}
        onInput={(e) => setUsername(e.target.value)}/>
    </label>
    <button type="button"
      onClick={() => onLogin(username)}>Login</button>
  </form>);
}
```

# Where should you `useState()`?

- Generally, declare that the "nearest common ancestor" of all Components that need that state

```
stateA is used by ComponentB and ComponentC

ComponentC is a "child" of ComponentD
ComponentB is a "child" of ComponentE
ComponentE is a "child" of ComponentD
```

ComponentD is the "nearest common ancestor"

- Have `useState()` for stateA in ComponentD
- Pass state and any setters/wrappers from ComponentD to child elements

# Often a LOT of state ends up at "top"

- Most state lives in App.jsx
    - Most state matters to most Components
- Temp state like "as you are typing" username
    - Kept out of App.jsx
    - Declared in their specialized components
    - Any "final" version passed to handlers received from ancestor
        - Ex: Login sends FINAL username to App
        - Using the `onLogin` prop it was passed

# Summary - State

- `import { useState } from 'react';`
- `useState()` is a React **hook**
- pass `useState()` initial value for a state variable
- returns array of two parts
    - We **destucture** array into two variables
    - state variable
    - setter function
- state variable will be:
    - Last value passed to setter function
    - Passed initial value if setter was never called

# Summary - Changing State

- Component returns HTML based on state
    - **conditional rendering**
- Can have multiple `useState()` calls
    - Each a different state variable
- When state changes, component **rerenders**
- set `onEVENT` (onClick, onSubmit, etc) props
    - If set on "native" HTML element
        - Callback called when event on element
    - Callback can call setter to change state

# Summary - Passing State

A Component

- Can pass state as props to other components
- CANNOT call setter functions they don't have
- CAN be passed functions as props
- CAN pass setter functions to other components
- CAN pass wrapper functions to other components