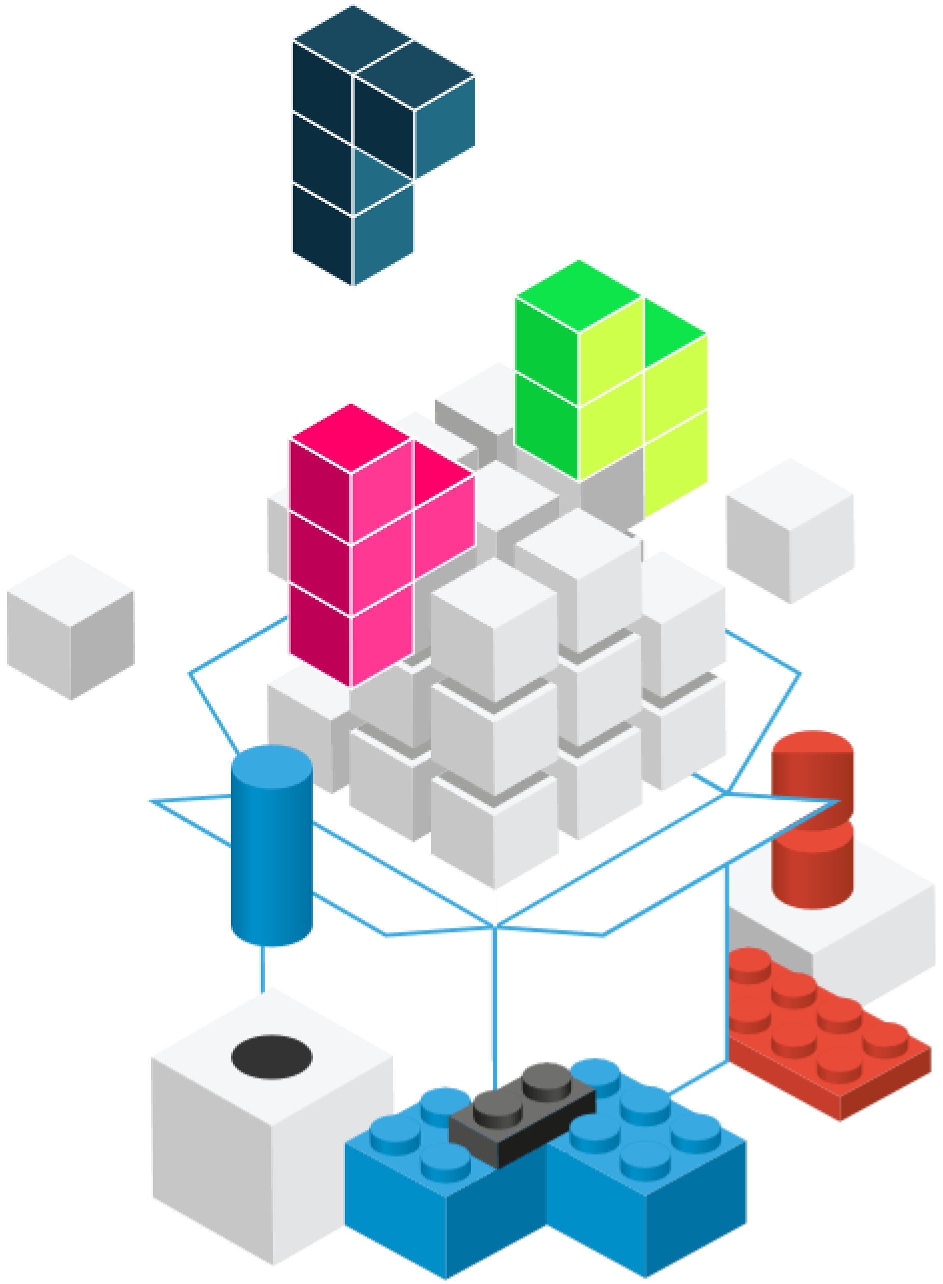
<WA1/>
<AW1/>
2025

Components and State

The Foundations of React

Fulvio Corno
Luigi De Russis



Outline

- React Hooks
- React Components
  - Props and State
  - The useState hook
- React design process
  - Top-down information flow

Full Stack React, “Appendix C: React Hooks”

React Handbook, chapter “Hooks”

https://react.dev/reference/react

Supercharge function components

HOOKS

Limitations of Function Components

- Simple
- Pure function (props->render)
- No state
- No side effects
- No lifecycle
- May define handler functions (not very useful, in absence of state)

Hooks

- Proposed in October 2018 – https://youtu.be/dpw9EHDh2bM
  - Stable since React 16.8 (February 2019), new hooks added almost in every version

- Additions to function components to access advanced features
  - Special mechanism for overcoming some limitations of “pure” functions, in a controlled way
  - Managing state, accessing external resources, having side-effects, …
- One *hook* call for each requested functionality
  - Hooks = special functions called by function components

Most Popular Hooks

Hook	Purpose
`useState`	Define a state variable in the component
`useEffect`	Define a side-effect during the component lifecycle
`useContext`	Act as a context consumer for the current component
`useReducer`	Alternative to useState for Redux-like architectures or complex state logic
`useMemo`	“Memoizes” a value (stores the result of a function and recomputes it only if parameters change)
`useCallback`	Creates a callback function whose value is memoized
`useRef`	Access to childrens’ ref properties
`useLayoutEffect`	Like useEffect, but runs after DOM mutations
`useDebugValue`	Shows a value in the React Developer Tools

https://react.dev/reference/react

https://react.dev/learn/describing-the-ui

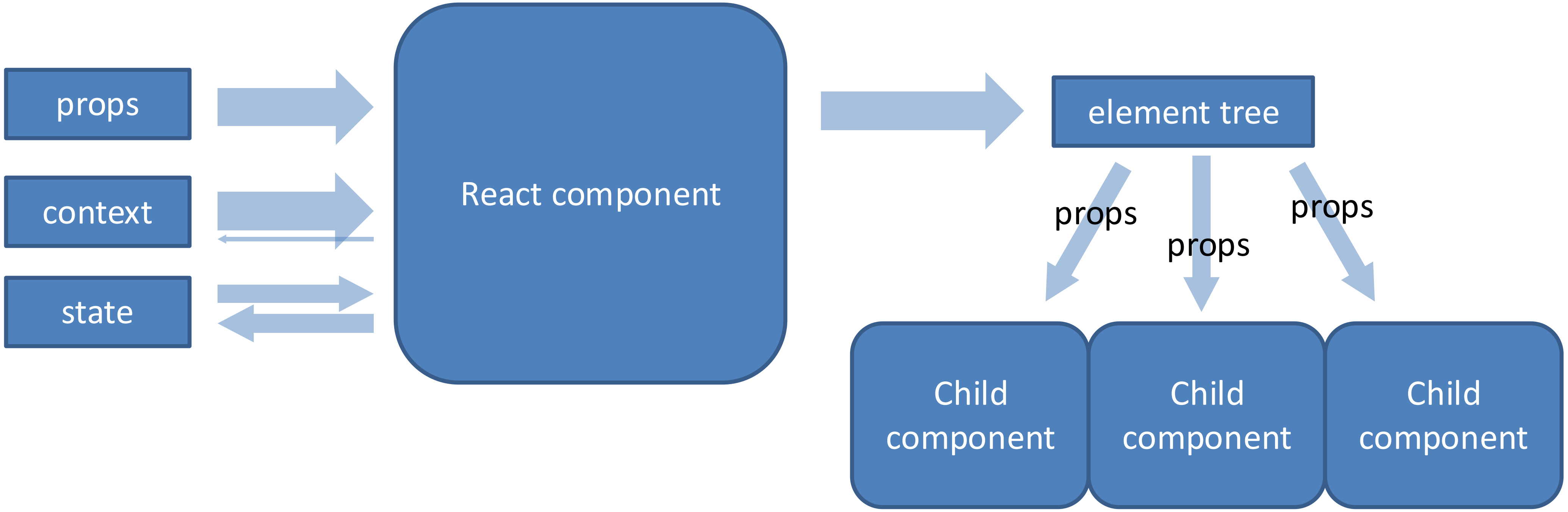
Full Stack React, Chapter “Advanced Component Configuration with props, state, and children” and “Appendix C: React Hooks”

React Handbook, Chapters “Props”, “State”, and “Hooks”

https://react.dev/learn/managing-state

COMPONENTS: PROPS AND STATE

Props, State, Context



Props, State, Context

- **Props** are immutable pieces of data that are passed into child components from parents
- **State** is where a component holds data, locally
  – When state changes, usually the component needs to be re-rendered
  – State is *private* to the component and is *mutable* from inside the component, only
- **Context** is a sort of "global" and "implicit" props, that are automatically passed to all interested components (later in the course…)

Passing Props

- In JSX, every attribute is converted to a prop
  - `<Header headerText='Hello'/>`
  - `props.headerText` will contain the string `"hello"`
- `props` is the argument of the Component Function and collects all passed props
  - They are all read-only
- May be any JS object, or other React elements
  - `<UserError level={3}/>`
  - `<ResultsTable displayData={latestResults}/>`

State

- An object containing local data, private to a component, that may be mutated by the component itself
- To define a state variable, use the `useState` hook

useState

- Creates a new state variable
  - Usually, a “simple” value
  - May be an object
  - Does not need to represent the whole complete component state
- It returns
  - A reference to the current value
  - A function to update the state value
- Update
  - With the new value
  - With a callback function

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

function ShortText(props) {
 const [hidden, setHidden] = useState(true);
 return (
    <span>
      {hidden ?
        `${props.text.substr(0, props.
maxLength)}...` : props.text }
      {hidden ? (
        <a onClick={() => setHidden(false)}>more</a>
      ) : (
        <a onClick={() => setHidden(true)}>less</a>
      )}
    </span>
  );
}
```

https://daveceddia.com/usestate-hook-examples/

Creating a State Variable

- `import{ useState } from 'react';`
- `const [hidden, setHidden] = useState(true);`
  - Creates a new state variable
  - `hidden`: name of the variable
  - `setHidden`: update function
  - `true`: default (initial) value
  - Array destructuring assignment to assign 2 values at once

- Creates a state variable of any type
  - **Remembered** across function calls!
- The default value sets the initial *value* (and *type*)
- The variable name can be used inside the function (to affect rendering)
- The `setVariable`() function will replace the current state with the new one
  - And trigger a re-render

Example

```
function WelcomeButton(props) {
  let [english, setEnglish] =
    useState(true) ;

  return (<button>
      {english ? 'Hello' : 'Ciao'}
    </button>) ;
}
```

- Call `useState` with the *initial version* of an object describing the component state
- Inside the component, you may refer the state variable to customize the result according to the current state

Updating the State

- **All** modifications to the state must be requested through set*Variable*(*newValue*)
- Never n-e-v-e-r modify the state variable directly
  - Always use the set*Variable* function
- It will apply the modification asynchronously (not immediately)

Updating the State

- With a new value
  - Dependent on props and constant values
  - Will **replace** the current one
  - Should have the same type (for consistent rendering)

```
setHidden(false) ;
```

- With a function

  `(oldState) => { return newState; }`
  - Executed as a callback
  - When the *new state depends on the old state*
  - The function return value will **replace** the current state
    - Must return a **new** state value
    - Must **not** mutate the passed-in state

```
setSteps(oldSteps => oldSteps + 1);
```

Function or Object in setVariable?

- If the logic for computing the next state depends on the current state, **always** use a function
- ✗ `setCounter(counter+1)`
  - `counter` is evaluated when `setCounter` **is called**
  - The new state will be assigned later, asynchronously
  - In case many asynchronous requests are made, some update may rely on out-of-date information
- ✓ `setCounter((cnt)=>(cnt+1))`
  - The arrow function will be evaluated when the async call is made, with an up-to-date value of `cnt`: guaranteed to have the latest value

https://medium.com/@wisecobbler/using-a-function-in-setstate-instead-of-an-object-1f5cfd6e55d1

Calling State Changes

- State changes are usually determined by asynchronous events
  - DOM event handlers
  - Server responses (e.g., API calls)
- The event handler is a function that in turn calls set*Variable*

```
function WelcomeButton(props) {
  let [english, setEnglish] =
    useState(true) ;

    const toggleLanguage = () => {
      setEnglish( e => !e ) ;
    }

  return (<button onClick={toggleLanguage}>
      {english ? 'Hello' : 'Ciao'}
    </button>);
}
```

Calling State Changes

- State changes are usually determined by asynchronous events
  - DOM event handlers
  - Server responses (e.g., API calls)
- The event handler is a function that in turn calls set*Variable*
  - Often implemented as an arrow function

```
function WelcomeButton(props) {
  let [english, setEnglish] =
    useState(true) ;

  return (<button
    onClick={()=>setEnglish((eng)=>(!eng))}>
      {english ? 'Hello' : 'Ciao'}
    </button>);
}
```

The default value

- Used during the first render of the component, only
  - Never used in successive renders
- May be a value, or a function
  - The function is called only during the initial render
- May be computed from the props
  - But will not update if the props change (beware bugs!)
  - Not recommended

Example

```
function Counter(props) {
  const [count, setCount] = useState(props.initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(props.initialCount)}>Reset</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

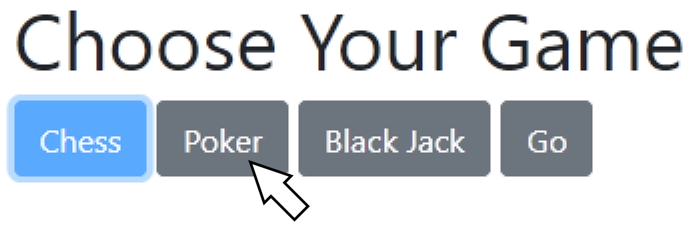
Multiple State Variables

- Do not use a single object for holding many (unrelated) properties
- Create as many state variables as needed, they are all independent
- Component will re-render if any state changes
- Children components will re-render only if *their* props change

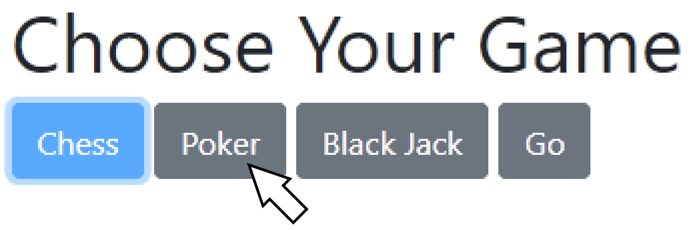
```
function Example(props) {

  [hidden, setHidden] = useState(true) ;
  [count, setCount] = useState(0) ;
  [mode, setMode] = useState('view') ;

  . . .
  setHidden(false) ;
  . . .
  setCount( c => c+1 ) ;
  . . .
  setMode('edit') ;
  . . .
}
```

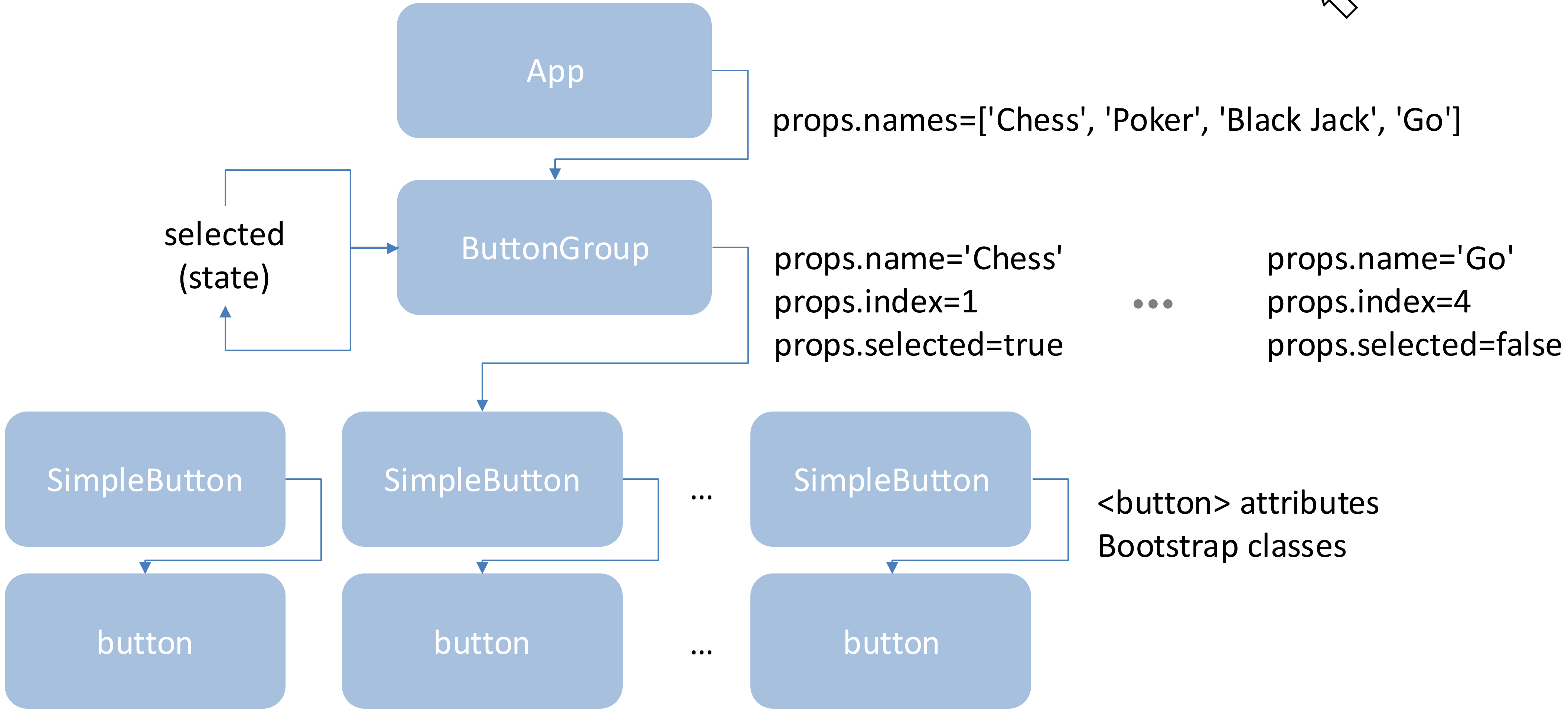
Can Children Mutate Parent's State?

- Each button may be selected or not, but only one may be selected at a time
- The information about what button is selected may not be in the button
- It is a state of a container component for "button group"



Analysis

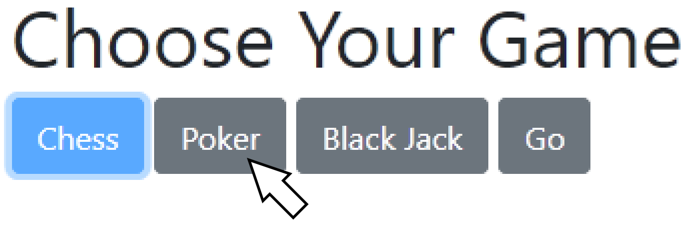

Choose Your Game
Chess
Poker
Black Jack
Go


App
props.names=['Chess', 'Poker', 'Black Jack', 'Go']
selected
(state)
ButtonGroup
props.name='Chess'
props.index=1
props.selected=true
•••
props.name='Go'
props.index=4
props.selected=false
SimpleButton
SimpleButton
...
SimpleButton
<button> attributes
Bootstrap classes
button
button
...
button

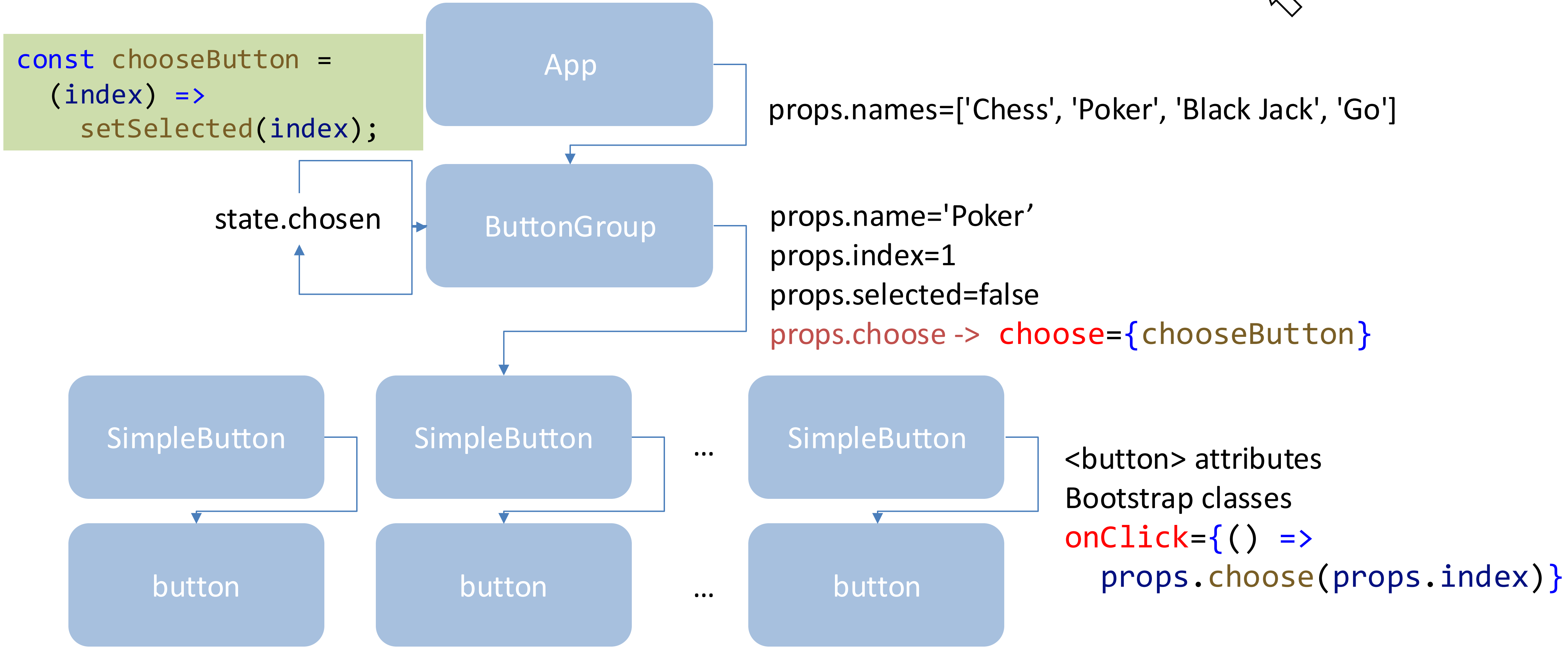
How To Change The Chosen Button?

- Handle `onClick` event from the button
- ButtonGroup must offer a method for changing the chosen option
  - will call `setSelected()`
- The method reference must be passed down to SimpleButton, with all other props

A Possible Solution



```
const chooseButton =
  (index) =>
    setSelected(index);
```



props.names=['Chess', 'Poker', 'Black Jack', 'Go']

props.name='Poker'
props.index=1
props.selected=false
props.choose -> `choose={chooseButton}`

<button> attributes
Bootstrap classes
```
onClick={() =>
  props.choose(props.index)}
```

React Design Hints

- Try to implement *stateless* components instead of *stateful* ones
  - Stateless components are more reusable
  - Stateless components are faster to execute
- Move *state* to common ancestors (“state lifting”)
- Pass *state* down to the children using *props*
- Allow children to ask for state updates, by passing down callback functions

License