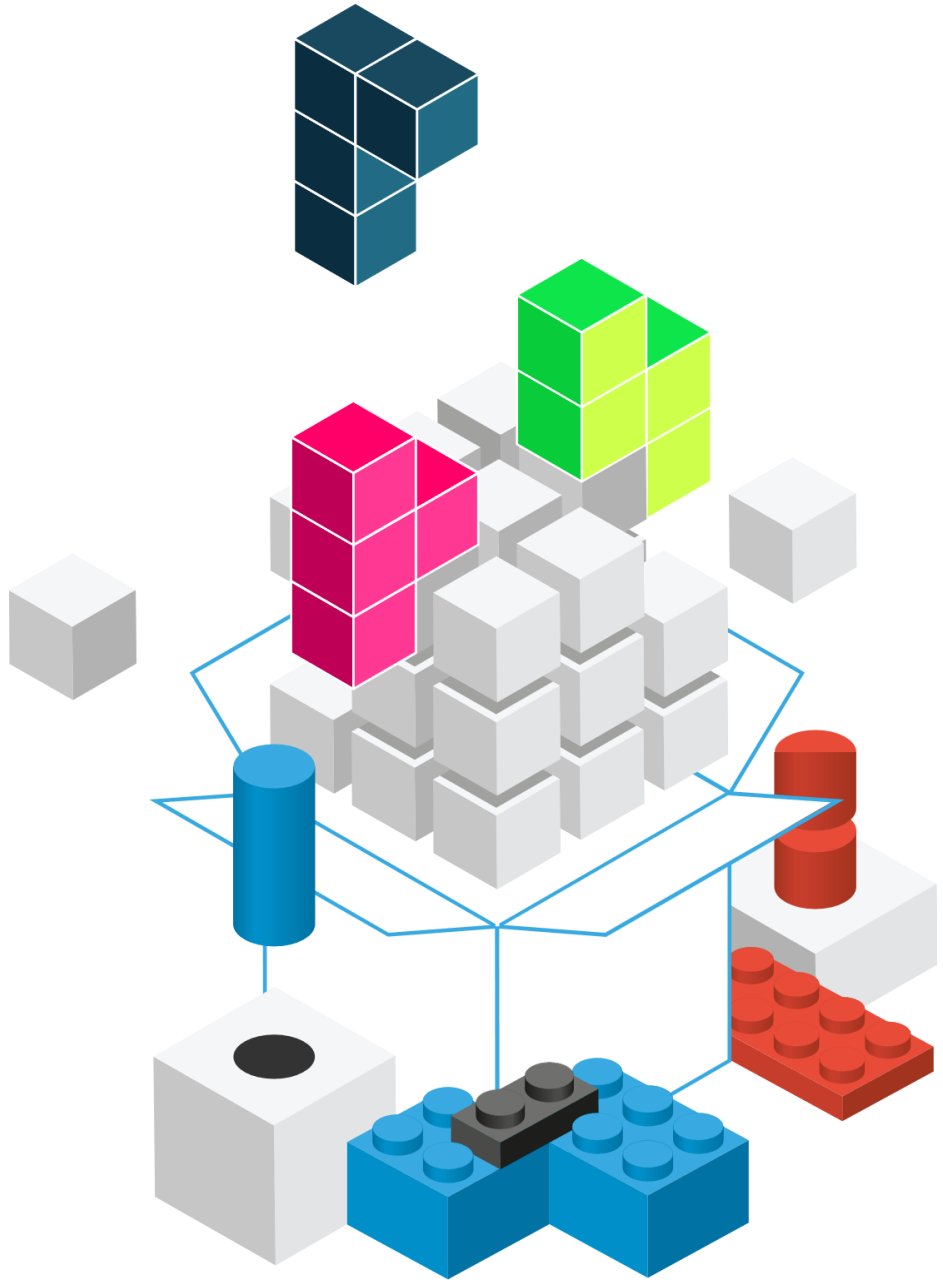
<WA1/>

2020

Components, Elements, JSX

The Foundations of React

Enrico Masala
Fulvio Corno
Luigi De Russis



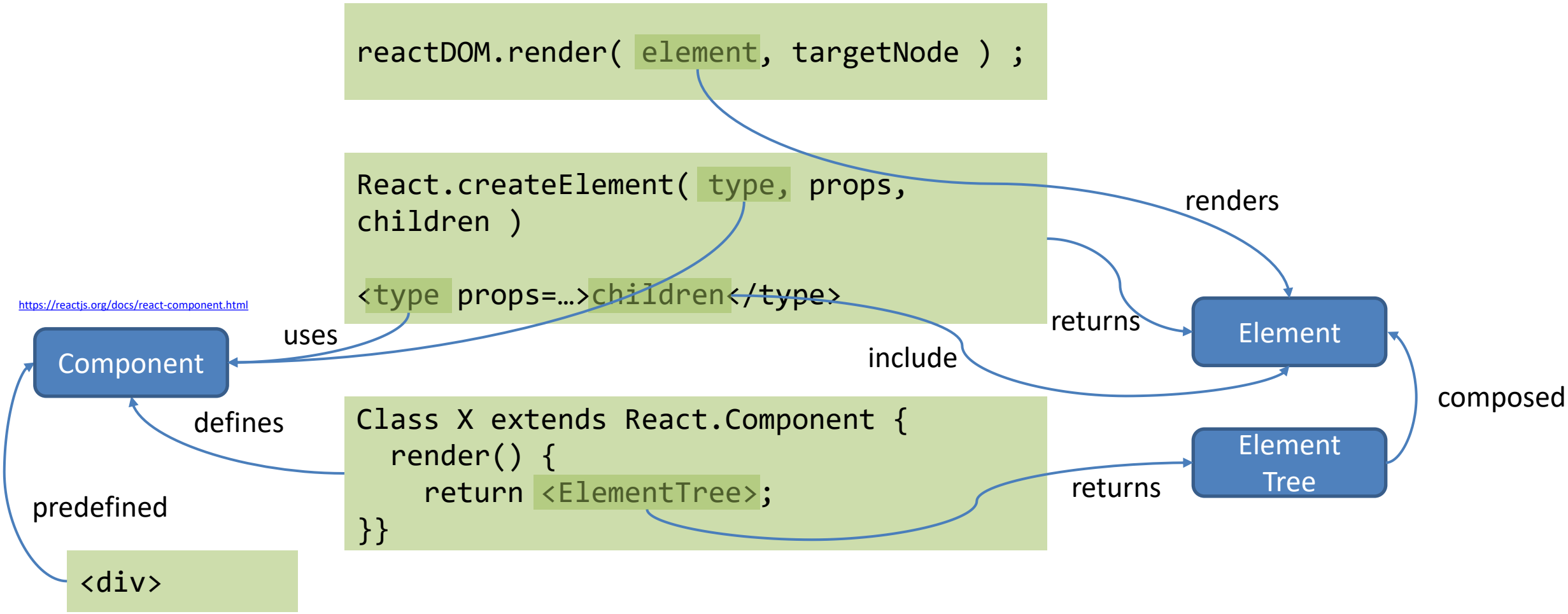
Outline

- React Elements
  - Creating
  - JSX language
- React Components
  - Defining
  - Props and State
  - Lifecycle
- React design process
  - Top-down information flow
  - Defining state
  - Adding Reverse flow

Part 1

Part 2

Conceptual Overview



https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html

Full Stack React, Chapter “JSX and the Virtual DOM”

Building block for describing web page content

REACT ELEMENTS

React Element

- An element is a plain object describing a component instance or DOM node and its desired properties
- A ReactElement is a representation of a DOM element in the Virtual DOM.
- It contains only information about
  - the component type (for example, a Button)
  - its properties (for example, its color)
  - any child elements inside it.
- Not an *instance* of a part of a page, but a *description* about how to construct it.
- Elements offer no callable methods.

React.createElement (1/3)

- `React.createElement( type, props, children )`

- Type
  - String: a DOM node identified by the tag name (e.g., `'div'`)
  - React component class/function: a user-defined component

React.createElement (2/3)

- `React.createElement( type, props, children )`

- Props: a simple object {}, containing:
  - DOM attributes for DOM nodes (type, src, href, alt, …)
  - Arbitrary values for React components (even array- or object-valued)
    - Available as `this.props` in the Component body
  - Represented as object properties (not strings like HTML attributes)
    - Exceptions (reserved words): `class` → `className`, `for` → `htmlFor`

React.createElement (3/3)

- `React.createElement( type, props, children )`

- Children:
  - a ReactNode object, that may be:
    - A string or number: text content of the nodes
    - A ReactElement (that may contain a tree of Elements)
    - An array of ReactNodes
  - nested Elements to be rendered as children of the element

Element Objects

Element with DOM nodes

```
{
  type: 'button',
  props: {
    className: 'button button-blue',
    children: {
      type: 'b',
      props: {
        children: 'OK!'
      }
    }
  }
}
```

Elements with React Component

```
{
  type: Button,
  props: {
    color: 'blue',
    children: 'OK!'
  }
}
```

Note: we don't create elements as plain objects. We always use createElement or JSX

Rendering Element objects

Element with DOM nodes

```
{
  type: 'button',
  props: {
    className: 'button button-blue',
    children: {
      type: 'b',
      props: {
        children: 'OK!'
      }
    }
  }
}
```

```
<button class='button
button-blue'>
  <b>
    OK!
  </b>
</button>
```

Elements with React Component

```
{
  type: Button,
  props: {
    color: 'blue',
    ren: 'OK!'
```

Rendering Element objects

Element with DOM nodes

```
{
  type: 'button',
  props: {
    className: 'button button-blue',
    children: {
      type: 'b'
      props:
        child
      }
    }
  }
}
```

- Components encapsulate element trees (generated given their properties).
- React asks the Button component to render itself. It will generate a tree of elements, to replace this one.
- ↻ Repeat until only DOM nodes are present.

Elements with React Component

```
{
  type: Button,
  props: {
    color: 'blue',
    children: 'OK!'
  }
}
```

```
{
  type: 'button',
  props: {
    className: 'button
button-blue',
    children: {
      type: 'b',
      props: {
        children: 'OK!'
      }
    }
  }
}
```

Conventions

- DOM Elements are always lowercase
  - `div p li img …`
- React Components are always uppercase
  - `WarningButton LoginForm TaskList …`
- The two types of elements can be mixed, nested, combined in any way
  - React uses *composition* and not *inheritance*
- Element trees describe **portions of the Virtual DOM**

https://reactjs.org/docs/jsx-in-depth.html

Full Stack React, Chapter “JSX and the Virtual DOM”

React Handbook, Chapter “JSX”

A humane way of describing trees of ReactElements

JSX

JSX – JavaScript Syntax Extension

- Alternative syntax for React.createElement
- XML fragments inside the JS code
  - Syntax details: all tags must be </closed> or <selfclosing/>
- Transpiled by Babel into plain JS

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

Element/Component name
Props
Children / Text content


```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
) ;
```

JSX Syntax

- May use <tag>…</tag> or <tag/> anywhere a JS expression is syntactically valid
  - Not only in Components
  - May also store in Arrays/Objects
  - After all, they are just ReactElements generated by React.createElement!
- May enclose in (…) for clarity

```
const element = <div className="main">Hello world</div>;
```

```
const element2 =  (<Message text="Hello world" />);
```

Note: use <tag/> if the component doesn't have any children

JSX Tag Name

- <Foo> is just React.createElement(Foo,…)
  - Foo must be in scope (imported or declared)
  - React must be in scope (even if it’s not visible in the code)

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
  return <CustomButton color="red" />;
}
```

Computed Tag Name

- The JSX Tag must be a valid identifier, can't be an expression
- If you want to select a component based on an expression, first assign it to a *capitalized* variable.

```
function WarningButton(props) {
  if(props.urgent)
    return <UrgentButton />;
  else
    return <NormalButton />;
}
```

```
function WarningButton(props) {
  const ThisButton = props.urgent ?
    UrgentButton : NormalButton ;
  return <ThisButton/>;
}
```

JSX Attribute Expressions

- Tag attributes are converted to props of the ReactElement
- String attributes become string-valued props
  - `color="blue" -> {color: 'blue'}`
- Other objects may be specified as a JS expression, enclosed in `{}`
  - `shadowSize={2} -> {shadowSize: 2}`
  - `log={true}`
  - `color={warningLevel === 'debug' ? 'gray' : 'red'}`
- Any JS expression is accepted

JSX Children

- The *content* between the tags <tag>*content*</tag> is passed as a special property `props.children`
- Such content may be:
  - A string literal
  - More JSX elements (nested components)

```
<MyComponent>Hello
world!</MyComponent>
```

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

  - Any {JS expression}
  - A {JS expression} returning an array of JSX elements (they are inserted as siblings)
  - A JS function (may be used as a callback by the Component)
  - Anything that the Component may understand (and render properly)

JSX Child Expressions

- JS expressions in { } may be used to specify element children
- One child (or an array of children) are generated by an expression

```
const Menu = (<ul>{loggedInUser ? <UserMenu /> : <LoginLink />}</ul>)
```

  - <JSX> inside {JS} inside <JSX> inside JS. Totally Legit.
- `undefined`, `null` or Booleans (`true`, `false`) are not rendered
  - Useful for conditionally including children

```
return (<ul>
  <li>Menu</li>
  {userLevel === 'admin' && renderAdminMenu()}
</ul>)
```

Boolean HTML Attributes in JSX

- In HTML some attributes do not have a value. Their simple presence "activates" a behavior
  - HTML: `<option value='WA' selected>Washington</option>`
  - HTML: `<input name='Name' disabled />`
- In JSX, a Boolean value must be given
  - True, for the presence of the attribute
  - False (or nothing) for the absence of the attribute
  - JSX: `<option value='WA' selected={true}>Washington</option>`
  - JSX : `<input name='Name' disabled={true} />`

Comments in JSX

- There are **no** comments in JSX
- The HTML/XML comments syntax `<!-- … -->` does **not** work
- If you want to insert comments, you must do that in an embedded JS expression (using JS syntax inside `{}`)

  `{/* … */}`

- Yes, it's ugly

DOM attribute names

- When passing props to a DOM native node, some differences exist
- Attribute names are camelCase
  - HTML onchange → JSX onChange
- The style attribute accepts an object and not a string
  - `<div style={{color: 'white'}}>Hello World!</div>`
  - Object keys are CSS Properties, and are camelCase (e.g., `margin-top` → `marginTop`)
  - Object values are CSS values, represented as strings

JSX Spread Syntax

- Shortcut syntax for passing all properties of an object as props to a React Component

```
const welcome = {msg: "Hello", recipient:
"World"} ;

<Component
  msg={welcome.msg}
  recipient={welcome.recipient} />
```

```
const welcome = {msg: "Hello", recipient:
"World"} ;

<Component {...welcome} />

// properties of the welcome object
// are “spread” as individual props
// with the same name
```

JSX Spread example (property passthrough)

```
const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "A" ? "ABtn" : "BBtn";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary"
        onClick={() => console.log("clicked!")}>
        Hello World!
      </Button>
    </div>
  );
};
```

- The ‘kind’ property is “consumed” by <Button>
- All other properties (…other) are passed to the child <button>
- In this way, <App> can specify the kind to Button and all other properties to “pass through” down the hierarchy

JSX Syntax Reminders

- The HTML `class` attribute is called `className`
- The HTML `for` attribute is called `htmlFor`
- HTML entities (< & © ☆ etc…) are not supported in JSX
  - Use the corresponding Unicode character (< & © ☆) inside a string in JS `{'☆'}`
  - Alternatively, use a Unicode Escape sequence: `{'\u2606'}`
    - See: https://www.toptal.com/designers/htmlarrows/

Data properties in DOM nodes

- React Components accepy *any property name* you need
  - `<Message level='urgent' code={123}/>`
- DOM nodes have a predefined set of properties
  - `<button value='Press me' ~~urgency='high'~~>…</button>`
- You may add new “custom” properties by prefixing their name with ‘`data-`’
  - `<button value='Press me' data-urgency='high'>…</button>`

https://reactjs.org/docs/components-and-props.html

https://reactjs.org/docs/react-component.html

https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html

Full Stack React, Chapter “Advanced Component Configuration with props, state, and children”

Putting together the building blocks

REACT COMPONENTS

Declaring components

Components as functions

```
const Button = ( props ) => (
  React.createElement( . . . )
     -or-
  <Element>...</Element>
);
```

Components as classes

```
class Button extends React.Component {
  render() {
    // receives this.props;
    return
      React.createElement( . . . ) ;
         -or-
      <Element>...</Element> ;
  }
}
```

Components:
Take props as their input
Return the elements as their output

Comparison

Components as functions

- Defined as function expressions or arrow expressions
- Receive (props) argument
- Must return a React Element tree
- The returned elements are function of the props
- Must be a pure function (no side-effects) and idempotent
- State and lifecycle may be managed with the recent Hooks, mechanism

Components as classes

- Must implement the render() method
  - Will return the React Element tree (same as function)
- Props are available as instance properties (this.props)
- May define other instance methods
- Local state may be defined
- Additional lifecycle events may be handled with custom logic

Top-Down Reconciliation

1. ReactDOM.render() or setState() is called for an element E
2. E() or E.render() is called, with the specified props, to obtain its representation as an Element tree ET (whose elements receive their props, as computed by E)
3. For each non-native element in ET, repeat steps 2-3 recursively
4. Stop when only native DOM elements are present in ET
5. The resulting ET is compared with the existing ET in the V-DOM, using an approximated $O(N)$ algorithm (https://reactjs.org/docs/reconciliation.html)
6. The differences are propagated to the browser's DOM

Tips for Creating Components

- It’s normal to create many different “small” components
- Each component is constructed by *composing* other components
  - Components may be repeated (with different props)
  - It’s up to the parent to determine the children’s props
- If a component becomes too complex, try to *extract small re-usable parts* as independent components

Lists and Keys (1/2)

```
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')
);
```

- The `NumberList` component generates a `<ul>` containing `<li>` for each of the numbers in its `props.numbers`
- Whenever you construct a list of elements, you should pass a unique **key** attribute to identify each item
- Unique keys help React identify which items have changed, are added, or are removed. They are used in the Reconciliation algorithm heuristics

https://reactjs.org/docs/lists-and-keys.html

Lists and Keys (2/2)

- Assign to each item in the list a special 'key' attribute, with unique values
  - `<li key={number.toString()}>{number}</li>`
- Most likely, we may reuse unique IDs from the data itself
  - `<li key={todo.id}>{todo.text}</li>`
- Keys must be specified when building the array of components
  - Usually in the `.map()` call, in the 'container' component
  - Not needed within the component of the items
- Uniqueness is only required within *the same list*
  - *Not globally* on the page
- Keys are *not* available as `props` in the component

React Fragments

- `render()` should always return a tree of elements, with a single root.
- To return a list of elements, you must include them in some "container" (such a)
  - This generates an "extra" DOM node, and in some contexts it might be invalid
- The special node `<React.Fragment>` may be used to wrap a list of element into a single root.
  - React.Fragment will not generate any node at the DOM level
- A shortcut syntax for fragments is `<>…</>`

https://reactjs.org/docs/fragments.html

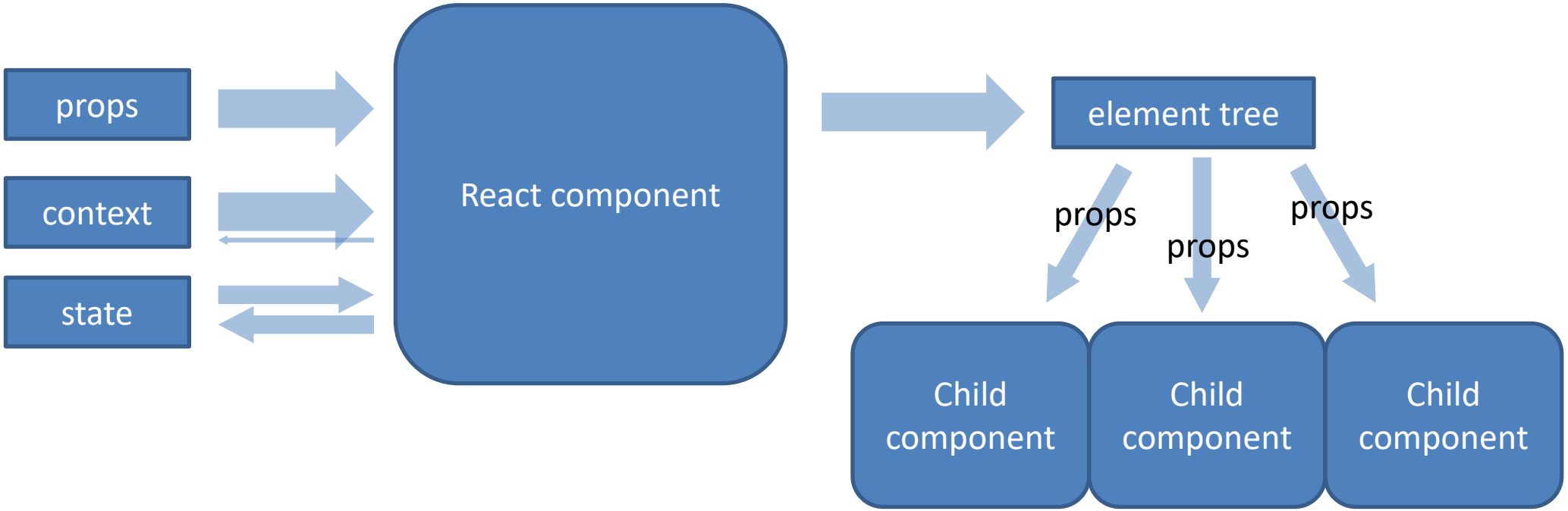
https://reactjs.org/docs/react-component.html

Full Stack React, Chapter “Advanced Component Configuration with props, state, and children”

React Handbook, Chapters “Props” and “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* by component methods
- Context is a sort of “global” and “implicit” props, that are automatically passed to all components

Passing props

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

PropTypes

- An *optional* mechanism for declaring the data types of props, that will be checked at run-time
  - npm install prop-types
- Create a static property `propTypes` in our component class
  - Maps every property name to a “type validator”
  - Validators check the props at run-time, only in development mode
  - Validation errors are shown in the JS console

https://reactjs.org/docs/typechecking-with-proptypes.html

PropTypes Example

```
import PropTypes from 'prop-types';

class Greeting extends React.Component
{
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

```
import PropTypes from 'prop-types';

class Greeting extends React.Component
{
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }

  static propTypes = {
    name: PropTypes.string
  };
}
```

PropTypes validators

- Optional props of pre-defined types
  - `PropTypes.array`
    - `PropTypes.arrayOf(PropTypes.number)`
  - `PropTypes.bool`
  - `PropTypes.funct`
  - `PropTypes.number`
  - `PropTypes.object`
    - `PropTypes.objectOf(PropTypes.number)`
    - `PropTypes.shape({…})`
  - `PropTypes.string`
- Rendering content
  - `PropTypes.node`: anything that can be rendered
  - `PropTypes.element`: any ReactElement
  - `PropTypes.MyComponent`
- Make a property required
  - `PropTypes.`*`type`*`.isRequired`
  - `PropTypes.any.isRequired`

Default Props Values

- We may specify default values
  - In case of missing properties
- Static property ‘defaultProps’
- If a prop is passed, the default is ignored
- Validation happens *after* default values are applied

```
class Counter extends React.Component {
  static defaultProps = {
    initialValue: 1
  };
  // ...
};
```

```
<Counter />
```

…equivalent to:

```
<Counter initialValue={1} />
```

State

- An object containing local data, private to a component, that may be mutated by the component itself
- To define state, simply assign an object to this.state in the class constructor
  - You need to define the component with the class syntax
  - You need to declare a constructor

State example

```
class WelcomeButton extends
React.Component {
  constructor(props) {
    super(props);
    this.state = { english: true };
  }

  render() {
    return <button>
      {this.state.english ?
          'Hello' : 'Ciao'}
    </button>
  }
}
```

- The constructor receives `props` and should call `super(props)`
- Assign to `this.state` the *initial version* of an object describing the component state
- Inside the component, you may refer to `this.state` to customize the result according to current state
- Never n-e-v-e-r modify `this.state` directly

State modifications

Merge: modifies (or adds) properties that you specify, leaves the others unchanged
Shallow: not recursive over nested objects

- `setState(updater, [callback])`
- **All** modifications to the state must be requested through setState()
- setState will apply the modification asynchronously (not immediately)
  - The callback will be executed after modification has been applied

- Updater as an object: it performs a *shallow merge* of the object's properties into the new state
  - `this.setState({num: 2});`
- Updater as a function
  - `(state, props) => updated_state`
  - Must return a **new** state object with the properties you want to modify
  - Must **not** mutate the passed-in state

https://reactjs.org/docs/react-component.html#setstate

Calling setState()

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

```
handleClick = () => {
  this.setState(
    {english: !this.state.english}
   );
};

render() {
  return <button
    onClick={this.handleClick}>
    {this.state.english ?
       'Hello' : 'Ciao'}
  </button>
}
```

Beware ‘this’: bind your functions

```
constructor(props) {
    super(props);
    this.state = { english: true };
    this.handleClick =
      this.handleClick.bind(this);
 }

 handleClick() {
    this.setState(
      {english: !this.state.english}
    );
 }
```

- When using class **methods** or function expressions, always bind the function in the constructor, or ‘this.state’ will fail
- When using **arrow** functions, it works because this is inherited from the class context

```
handleClick = () => {
    this.setState(
      {english: !this.state.english}
    );
};
```

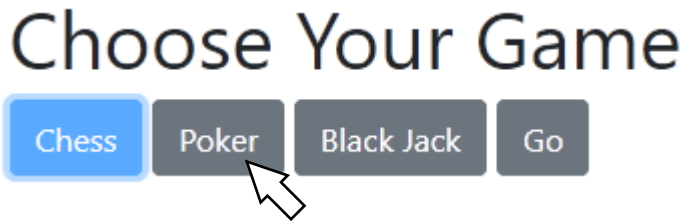
Function or Object in setState?

- If the logic for computing the next state depends on the current state, **always** use a function
- `setState({ counter: this.state.counter+1})`
  - `this.state.counter` is evaluated when `setState` **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
- `setState((state,props)=>({counter:state.counter+1}))`
  - `state.counter` will be evaluated when the async call is made: guaranteed to have the latest value

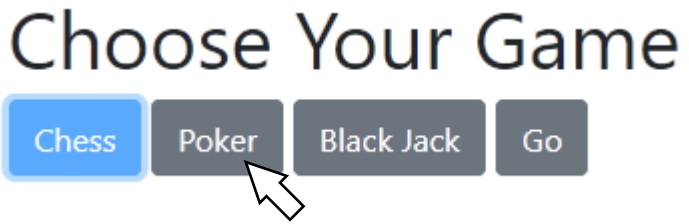
https://medium.com/@wisecobbler/using-a-function-in-setstate-instead-of-an-object-1f5cfd6e55d1

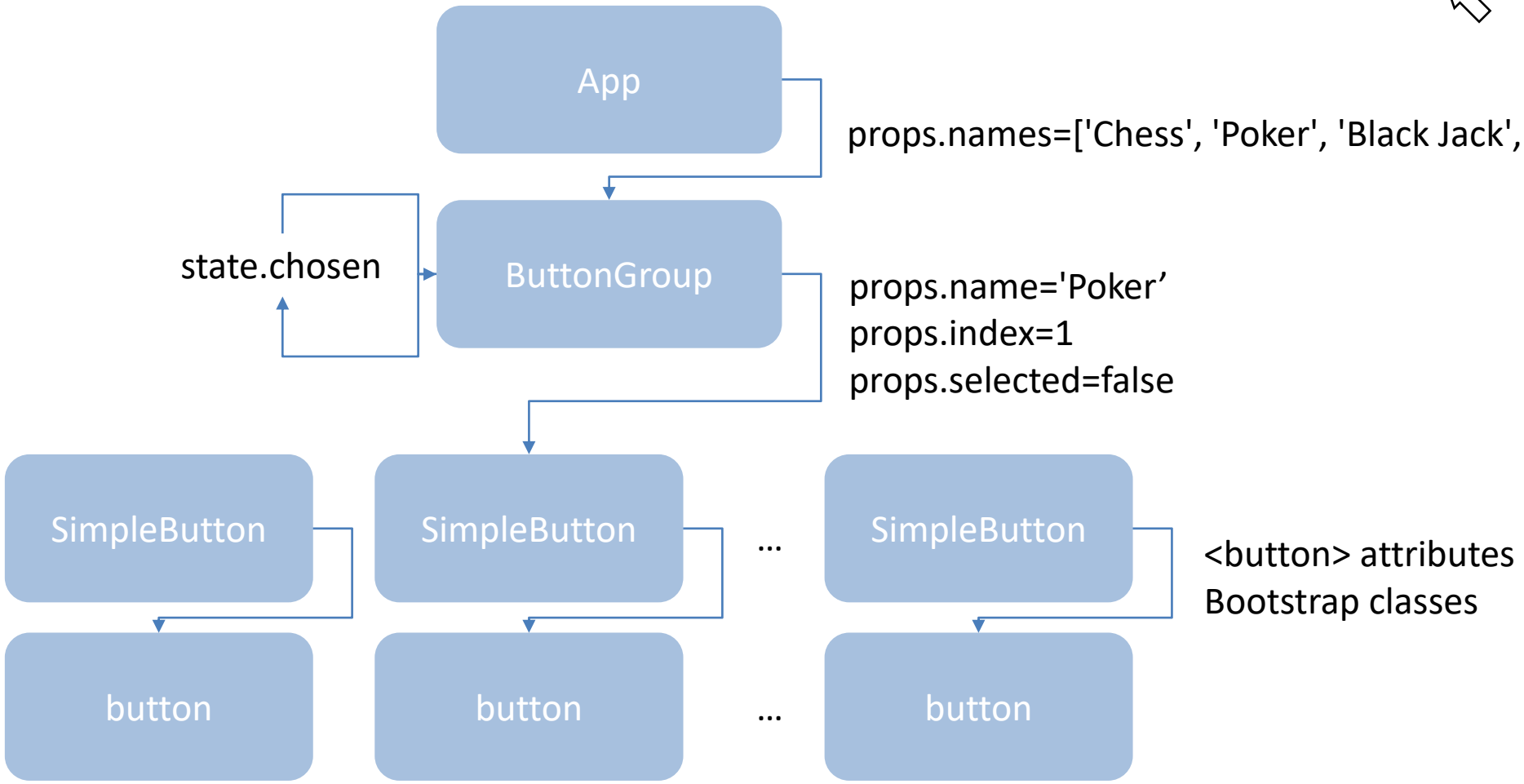
Can children mutate parent's state?

- 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

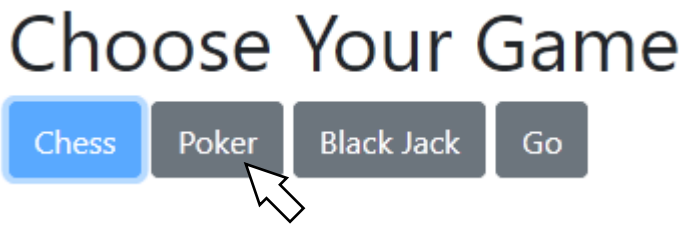

Choose Your Game
Chess
Poker
Black Jack
Go


App
props.names=['Chess', 'Poker', 'Black Jack', 'Go']
state.chosen
ButtonGroup
props.name='Poker'
props.index=1
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 setState()
- The method reference must be passed down to SimpleButton, with all other props

Solution




```
changeSelected = (id)=>{
  this.setState(
    {chosen: id});
}
```

state.chosen

App

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

ButtonGroup

props.name='Poker'
props.index=1
props.selected=false
props.changeSelected=this.changeSelected

SimpleButton

SimpleButton

...

SimpleButton

button

button

...

button

<button> attributes
Bootstrap classes
onClick={()=>props.changeSelected(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
  - Stateless components may be declared as a simple function
- 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

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/