BUILDING COMPONENTS

Agenda

- All practical details for building components
- Working with inputs

Camel case

- JSX is more JavaScript than HTML
- So it uses camelCase property naming
  - class → className
  - tabindex → tabIndex
- Are you ok with that ?

```
render() {
  return (
    <div className="App">
      <h1>Hello React</h1>
    </div>
  );
}
```

More like XML

- An element without content can be closed immediately
- Not like HTML ☹

```
render() {
  return <div className="App">
    <h1>Hello React</h1>
    <p/>
  </div>;
}
```

```
<div class="App">
 <h1>Hello React</h1>
 <p></p>
</div>
```

HTML is Escaped

- JSX is always escaped
  - Prevent XSS

```
<div className="App">
  {this.title}
</div>
```

```
<h1>Hello React</h1>
```

```
<div className="App">
  <div dangerouslySetInnerHTML={{__html: this.title}}></div>
</div>
```

Hello React

Ugly but by design

Functional Component

- The following is considered a component

```
export function ContactItem(props: ContactProps) {
  const {contact} = props;

  return <div className={styles.ContactItem}>
    <span>{contact.name}</span>
  </div>;
}
```

Must start with capital letter

- This is a powerful idiom
- However, component classes offers more features

Props are read only

- A component must never modify its own props
- Props are sent from the parent
- Therefore are "owned" by the parent

```
export class Counter extends React.Component<CounterProps> {
   constructor(props) {...}

   render() {...}

   dec = () => {
      this.props.count--;
   }
}
```

React will not allow that at runtime

Component Cleanup

- A component may allocate resources
  - Timer
  - WebSocket
  - Event Listeners
- Use componentWillUnmount to cleanup

Component Cleanup


```
export class Clock extends React.Component<{}, ClockState> {
  intervalId;

  constructor(props) {
    super(props);

    this.state = {
      time: new Date(),
    };
  }

  componentDidMount() {
    this.intervalId = setInterval(this.onTick, 1000);
  }

  componentWillUnmount() {
    clearInterval(this.intervalId);
  }

  render() {
    return <span>{this.state.time.toLocaleTimeString()}</span>
  }

  private onTick = () => {
    this.setState({
      time: new Date()
    });
  }
}
```

setState

- Do not modify state directly

```
private onTick = () => {
   this.state.time = new Date();
}
```

- React cannot detect this kind of change → render method is not invoked
- setState merges the specified object into the current state
  - Thus, setting one field does not reset the other

setState

- React may batch multiple setState calls

render is invoked only once ☺

However, the final counter is 1 and not 2 ☹

```
inc = ()=> {
  this.setState({
    counter: this.state.counter+1,
  });

  this.setState({
    counter: this.state.counter+1,
  });
}
```

setState – The right way


```
inc = () => {
  this.setState((state, props) => {
    return {
      counter: state.counter + 1,
    }
  });

  this.setState((state, props) => {
    return {
      counter: state.counter + 1,
    }
  });
}
```

Note that setState now receives a callback, not an object

Unidirectional Data Flow

- Any state is owned by a specific component
- A state change effects only the current/below components
- A user event captured by nested component must be propagated up
  - Without propagating the parent component will not be re-rendered
- But, how do we propagate the event ?
- What about forms ?

Events

- Named using camelCase
- Do not return false
  - Use e.preventDefault
- The this keyword does not point to the component instance ☹

this is undefined

```
render() {
   return <button onClick={this.run}>Click me</button>;
}

run() {
   console.log(this);
}
```

Synthetic Event

- An event handler receives a “fixed” event object
- React is responsible for normalizing the event according to the standard
  - Beware, event objects are pooled and reused !!!

```
run = (e) => {
   console.log(e == e.nativeEvent);
}
```

Always false

Self Hide

- A component might want to hide itself
- The render method is allowed to return null

```
export class Child extends React.Component {
   constructor(props) {
      super(props);
   }

   render() {
      return null;
   }
}
```

List & Keys

- Think about the following implementation

```
render() {
  const {contacts} = this.state;

  return <div className="App">
    <div>
      <button onClick={this.refresh}>Refresh</button>
    </div>

    <ul>
      {contacts.map(c => <li>
        <ContactItem contact={c}/>
      </li>)}
    </ul>
  </div>
}
```

```
refresh = ()=> {
  this.setState({
    contacts: randomSort([
      {"id": 1, "name": "Ori"},
      {"id": 2, "name": "Roni"},
      {"id": 3, "name": "Udi"},
      {"id": 4, "name": "Tommy"},
    ])
  });
}
```

What does happen when a contact moves inside the list ?

Key

- If a key is not specified React does not track list transformation
- Each component might receive new props
- If a component has internal state then it might be unsynchronized with the new props

```
render() {
  console.log("ContactItem.render");

  const {contact} = this.props;

  return <div>
    <span>{contact.name}</span>
    <button onClick={this.dec}>Dec</button>
    <span>{this.state.counter}</span>
    <button onClick={this.inc}>Inc</button>
  </div>
}
```

Set a key

- Using the key, React can associate new element with an old one
- Thus, each component may "move" with its original element

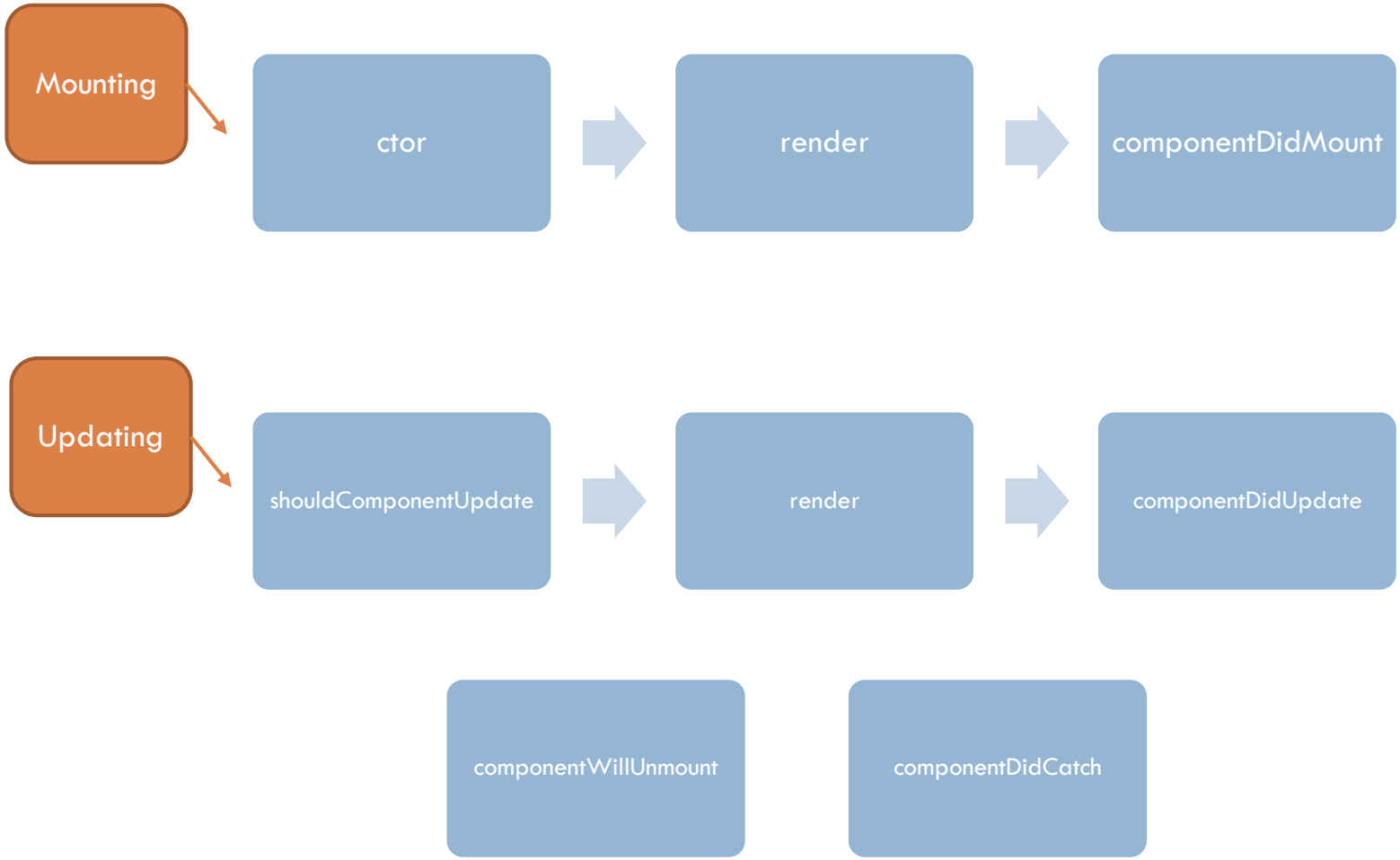
Think about it carefully !!!

```
render() {
  const {contacts} = this.state;

  return <div className="App">
    <div>
      <button onClick={this.refresh}>Refresh</button>
    </div>

    <ul>
      {contacts.map(c => <li key={c.id}>
        <ContactItem contact={c} onDelete={this.onDeleteContact}/>
      </li>)}
    </ul>
  </div>
}
```

Lifecycle methods



Pure Component

- A component that depends only on its props

Impure

```
render() {
  console.log("ContactItem.render");

  const {contact} = this.props;

  return <div>
    <span>{contact.name}</span>
    <button onClick={this.dec}>Dec</button>
    <span>{this.state.counter}</span>
    <button onClick={this.inc}>Inc</button>
    <button onClick={()=>this.props.onDelete(contact)}>Delete</button>
  </div>
}
```

- Pure components are usually easier to understand

Working with input

- input element has a value property
- A component has this.state
- Who is the source of truth ?

The user is unable to change the input

```
export class App extends React.Component<{}, AppState> {
  constructor(props) {
    super(props);

    this.state = {
      name: "Ori",
    }
  }

  render() {
    return <div className="App">
      <h1>Hello React</h1>

      <span>Name:</span>
      <input value={this.state.name} />
    </div>
  }
}
```

Two approaches

- Controlled component
  - Component's state is the source of truth
  - Need to manually synchronized input state with component state
- Uncontrolled component
  - Input's state is the source of truth
  - Need to manually update input state

Controlled Component

- A bit complicated when dealing with complex inputs that support multiple "change" events

```
render() {
   return <div className="App">
      <h1>Hello React</h1>

      <span>Name:</span>
      <input value={this.state.name} onChange={this.onInputChange} />
   </div>
}

onInputChange = (e) => {
   this.setState({
      name: e.target.value
   });
}
```

Uncontrolled Component


```
export class App extends React.Component<{}, AppState> {
  inputName: RefObject<HTMLInputElement>;

  render() {
    return <div className="App">
      <h1>Hello React</h1>

      <span>Name:</span>
      <input value={this.state.name} onChange={this.onInputChange} ref={this.inputName}/>

      <div>
        <button onClick={this.sayHello}>Say Hello</button>
        <button onClick={this.reset}>Reset</button>
      </div>
    </div>
  }
```

```
constructor(props) {
    super(props);

    this.state = {
      name: "Ori",
    }

    this.inputName = React.createRef();
  }
```

```
onInputChange = (e) => {
    this.setState({
      name: e.target.value
    });
  }

  sayHello = () => {
    alert("Hello " + this.inputName.current.value);
  }

  reset = () => {
    this.inputName.current.value = "";
  }
```

Formik

- A small library
- Helps dealing with forms
- Offers component
  - Fromik
  - Field
  - FieldArray
  - ErrorMessage
  - More …
- See https://jaredpalmer.com/formik/docs/overview

Containment

- A component might not know its children ahead of time. Think about
  - Dialog
  - Toolbar

```
<Dialog>
   <h2>Title</h2>

   <button>Close</button>
</Dialog>
```

```
render() {
   if(!this.props.children) {
      return null;
   }

   return <div className={styles.Dialog} onClick={this.onOverlayClick}>
      <div className={styles.content} onClick={this.onContentClick}>
         {this.props.children}
      </div>
   </div>
}
```

Multi Containment

- Use props to specify the content

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}
```

Inheritance

- Technically it is possible
- However, React doesn't like it
- Prefer composition
- Non UI functionality should be extracted into a separate module

Code Splitting

- A.K.A lazy loading
- Webpack looks for the import syntax
- New bundle is created automatically during build time
- Must not reference the lazy loaded code
  - This is tricky and error prone ☹
  - You will not get errors if lazy loaded code is already part of the main bundle

Code Splitting


```
export class App extends React.Component<{},
AppState> {
   constructor(props) {
      super(props);

      this.state = {
         Clock: null,
      }
   }

   render() {
      const {Clock} = this.state;

      return <div className="App">
         <h1>Hello React</h1>

         <button onClick={this.load}>Load</button>

         {Clock ? <Clock /> : null}
      </div>
   }
}
```

Magic starts here …

```
load = async () => {
      let {Clock} = this.state;
      if(!Clock){
         const clockModule = await import("./Clock");
         Clock = clockModule.Clock;

         this.setState(state => ({
            Clock,
         }));
      }
   }
```

Context

- Pass data through the component tree
- Middle components are not aware
- Start with defining the context

```
export const UserContext = React.createContext(null);
```

- Add a provider

```
<UserContext.Provider value={user}>
  <div className="App">
    <h1>React App</h1>

    <button onClick={this.logout}>Logout</button>

    <Home/>
  </div>
</UserContext.Provider>
```

Dynamically set context value

Context

- Consume it every where

```
export class UserStatus extends React.Component {
   constructor(props) {
      super(props);
   }

   render() {
      const now = new Date();

      return <UserContext.Consumer>
         {
            user => (user ? <span>Hello, {user.name}</span> : null)
         }
      </UserContext.Consumer>

   }
}
```

Error Boundary

- A component may implement componentDidCatch
- Will be invoked in case of an error during rendering of child component
  - Not self render !!!

```
export class ErrorBoundary extends React.Component {
  componentDidCatch(error, info) {
    console.log("componentDidCatch", error, info);
  }

  render() {
    //
    //  In case of an error can render fallback UI
    //
    return this.props.children;
  }
}
```

Fragment

- Group a list of elements without adding extra node

```
export class App extends React.Component {
   constructor(props) {
      super(props);
   }

   render() {
      return <>
         <h1>Hello React</h1>
         <h2>How are you</h2>
      </>
   }
}
```

No node is created. Can also use the React.Fragment syntax

Higher Order Component

- A.K.A HOC
- A function that takes a component and returns a new one

```
export const appStore: AppStore = {
   user: {
      id:1,
      name:"Ori",
   }
}

export function connectToStore(component: any) {
   return () => React.createElement(component, appStore);
}
```

Portal

- Allows a component to render any where inside the DOM

```
export class Modal extends React.Component {
  el: HTMLElement;
  root: HTMLElement;

  constructor(props) {
    super(props);

    this.root = document.querySelector("body");
    this.el = document.createElement('div');
  }

  componentDidMount() {
    this.root.appendChild(this.el);
  }

  componentWillUnmount() {
    this.root.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(this.props.children, this.el);
  }
}
```

Summary

- Functional component is cool
- props are immutable
- Component should render according to state/props
- A component may raise an event and thus let its parent re-render it self