Context, Life Cycle, Forms

Making React Components Alive

Enrico Masala

Fulvio Corno

Luigi De Russis
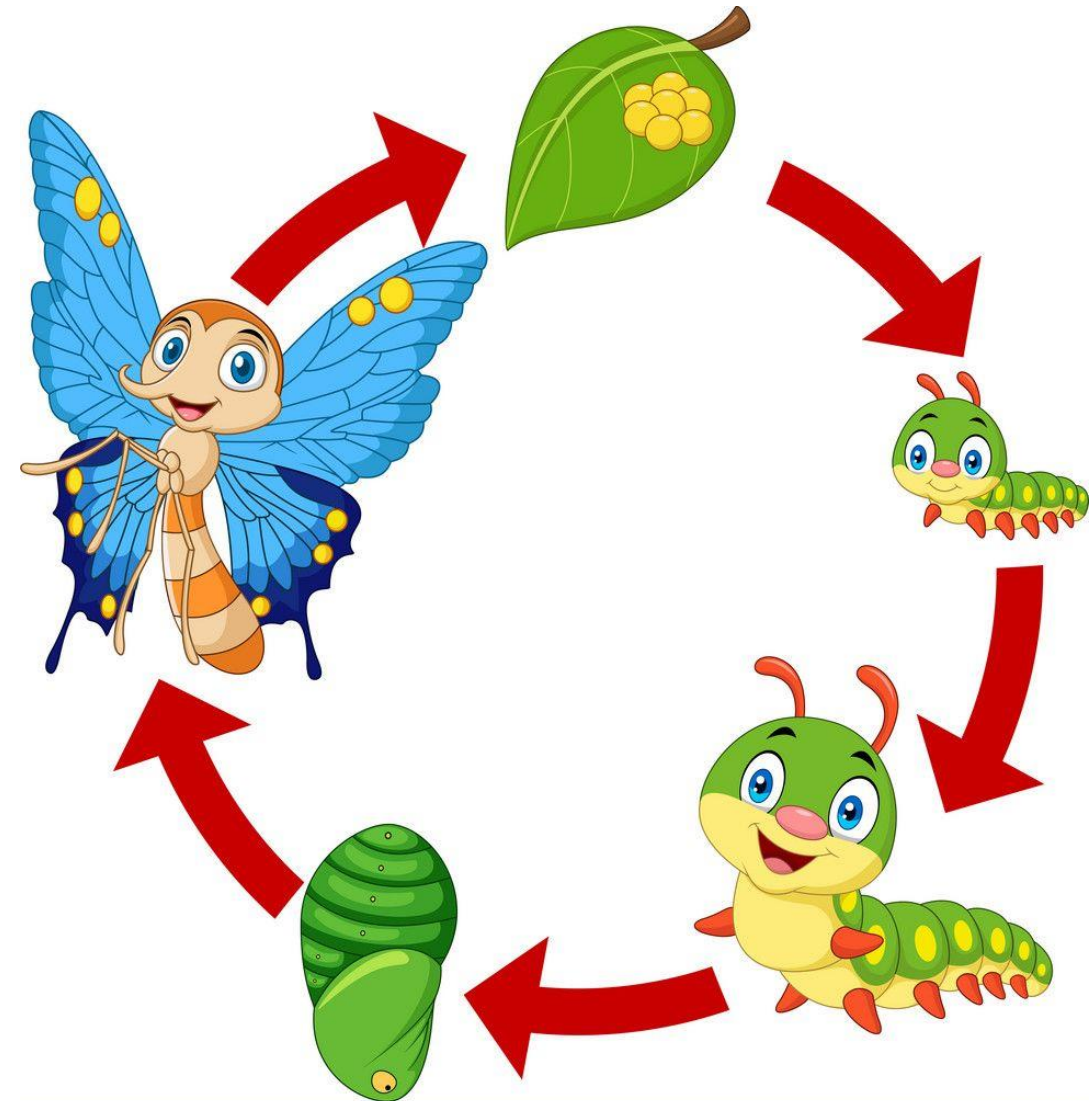
POLITECNICO
DI TORINO

Outline

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

Part 1 ✓

Part 2

Sort-of Globally Available Props (to avoid props drilling)

CONTEXT

Context

Unidirectional information flow +

Functional components =

Must pass every prop to the component that needs it, and sometimes it means "drilling through" many components with several props

- Solution: the Context API offers a "global" set of props that are "automatically" available to lower components
  - Without declaring them explicitly at every level
- "Props teleporting"

Examples

- The current visual theme for the whole page (e.g., dark, light, ...)
  - Needed by most visual components (towards the bottom of the tree)
  - Not needed by any container component
- Logged in/logged out status (and basic user information)
  - Needed to enable/disable large portions of the page
  - Needed to provide user info in various parts of the page (e.g., avatar)
  - Needed to call remote APIs with user-related queries
- Shared data cache

Context Ingredients

- Context definition
  - const *ExampleContext* = `React.createContext`()
  - Defines a context object with a `name`
- Context provider
  - <*ExampleContext*.`Provider value`=...> component
  - Injects the context `value` into all nested components
- Context consumer
  - `MyComponent`.`contextType` = *ExampleContext*
    - Context value available in `this.context`
  - <*ExampleContext*.`Consumer`>
    - Renders a function that receives the context `value` as a parameter

Context Definition

- const *ExContext* = `React.createContext`(defaultValue)
- Creates a new Context object
  - *ExContext*.`Provider` and *ExContext*.`Consumer`
  - Contains the value of one object
  - The *ExContext* identifier is used in value propagation
- Components may subscribe (consume) to this context
  - The provided value comes from the closest *Provider* ancestor
  - If no provider is found, the defaultValue is used
  - In all other cases, defaultValue is ignored

Context Provider

- A component *ExContext*.`Provider` is automatically created for each new Context

- The component specifies a `value` prop, that is available to all nested "consumer" components (even if deeply nested)
  - Consumers MUST be nested inside the provider
  - Providers may be anywhere (assuming the context object is visible)

- Providers may be nested: each level may override the previous `value`

- When the Provider's `value` changes, all consumers will re-render

Context Consumer (function or class component)

- The component *<ExContext.Consumer>* may be used in the render function/method

- You must provide a *callback function* that
  - Receives the context value (from the closest provider, or defaultValue if not provider is found)
  - Returns the React Element to be rendered

```
<ExContext.Consumer>
  {value => /* render something
              based on the context value */}
</ExContext.Consumer>
```

Context Consumer (class component)

- You may add a class property `contextType` to any consumer component (defined with the class syntax)

- It creates the property `this.context` that contains the (closest) provider's value
  - May be used in all component's methods

```
class MyComponent extends React.Component {
  render() {
      ... Use this.context ...
  }
}
MyComponent.contextType = ExContext;
```

```
class MyComponent extends React.Component {
    static contextType = ExContext;
    render() {
        ... Use this.context ...
    }
}
```

Changing context values

- When a Consumer child needs to update the context value, the Provider must provide a function callback to perform the update
  - As a prop (by drilling the nesting levels)
  - As part of the context value
- Remember: the **state** is part of the component containing the Provider
  - Not in the provider itself
  - Not in the context object

Example

```
class Container extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      msg: 'hey'
    }
  }

  render() {
    return (
      <Ctx.Provider value={{
          state: this.state,
          updateMsg: () =>
            this.setState({msg: 'ho!'})
        }}>
        {this.props.children}
      </Ctx.Provider>
    )
  }
}
```

```
class Button extends React.Component {
  render() {
    return (
      <Ctx.Consumer>
        {(context) => (
          <button onClick={context.updateMsg}>
            {context.state.msg}</button>
        )}
      </Ctx.Consumer>
    );
  }
}

class HelloWorld extends React.Component {
  render() {
    return <Container>
            <Button />
           </Container> ;
  }
}
```

Caveats

- Don't put everything into Context
  - Defeats component portability
  - Reduces "purity" of functional components
- Don't use it for programming laziness
  - Explicit parameter passing is also a good documentation practice
- Don't use it to correct design errors
  - Often, a refactoring of the component tree (and props/state lifting) may be a cleaner solution

...before you consider Context

- Passing a component as a prop (inversion of control)
  - When a nested components needs many props from and upper component
  - The upper component defines JSX of the element (using the available info)
  - The component itself is passed as a prop (just one prop, that will be passed and rendered)
- Use "render props"
  - Callback functions as props (https://reactjs.org/docs/render-props.html)
  - The lower component will call the "render prop" at render time, that has access to the upper component's props and state
- Use Children Components

Children Components and props.children

- Every time we nest components in JSX, a special prop is added
  - `props.children`
    - A single element
    - A list of elements

- In the component, you may render `{this.props.children}` to include the nested elements

```
return (
  <Container>
    <Article headline="An interesting Article">
      Content Here
    </Article>
  </Container>
)
```

```
class Container extends React.Component {
  render() {
    return <div className="container">
      {this.props.children}
    </div>;
  }
```

Manipulate your children

- `this.children` is an iterable data structure (`React.Children`)
  - May use `.map()` to create a list of (modified) children
  - May use `.forEach()` to iterate and examine the children
- For ease of manipulation, we may use `.toArray()`
  - All array methods (e.g., `.sort()`) may be used
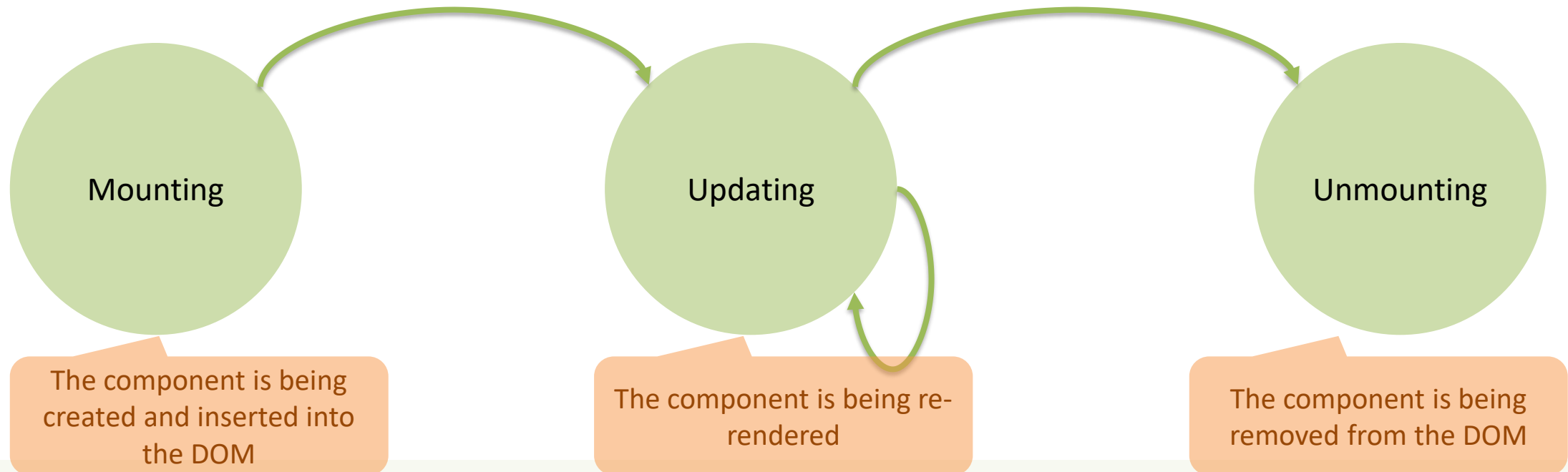- The children may be "customized" by the parent

https://reactjs.org/docs/react-api.html#reactchildren
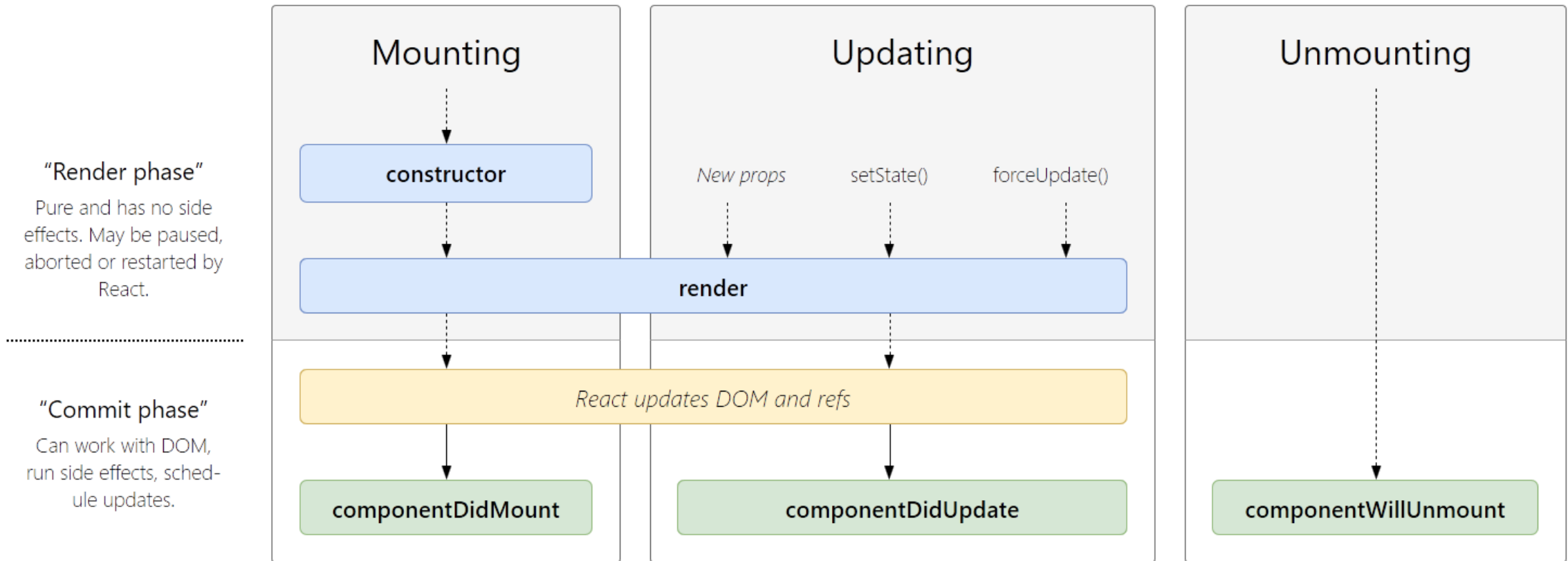
There's life before and after render()

COMPONENTS' LIFECYCLE

Lifecycle events

- The render() method is the most important action for a component
- Several other methods may be defined, to customize what happens at different moments in the evolution of the component

Mounting

Updating

Unmounting

The component is being created and inserted into the DOM

The component is being re-rendered

The component is being removed from the DOM

Lifecycle methods

http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

Mounting methods

- `constructor()`
  - `super(props)`
  - Initialize local state with
    `this.state = {…}`
    - Don't call `setState`
  - Bind event handler methods (if needed)
  - Should not have side-effects or subscriptions

- `render()`

- `componentDidMount()`
  - Invoked immediately after a component is mounted
  - Initialization that requires the existence of DOM nodes
  - *Load data* from a remote endpoint (REST API)
  - May call `setState` (triggers an extra render)

Updating/Unmounting methods

- `componentDidUpdate()`
  - Invoked immediately after updating occurs
  - Not called for the initial render
  - May launch network requests (only if props have changed!)
  - May call `setState`, but only conditionally (will cause infinite loop)

- `componentWillUnmount()`
  - invoked immediately before a component is unmounted and destroyed
  - Perform *cleanup* (timers, pending network requests, subscriptions...)
  - Don't call `setState`
  - Once a component instance is unmounted, it will never be mounted again
    - Possibly, a new instance will be created

Lifecycle methods (full)
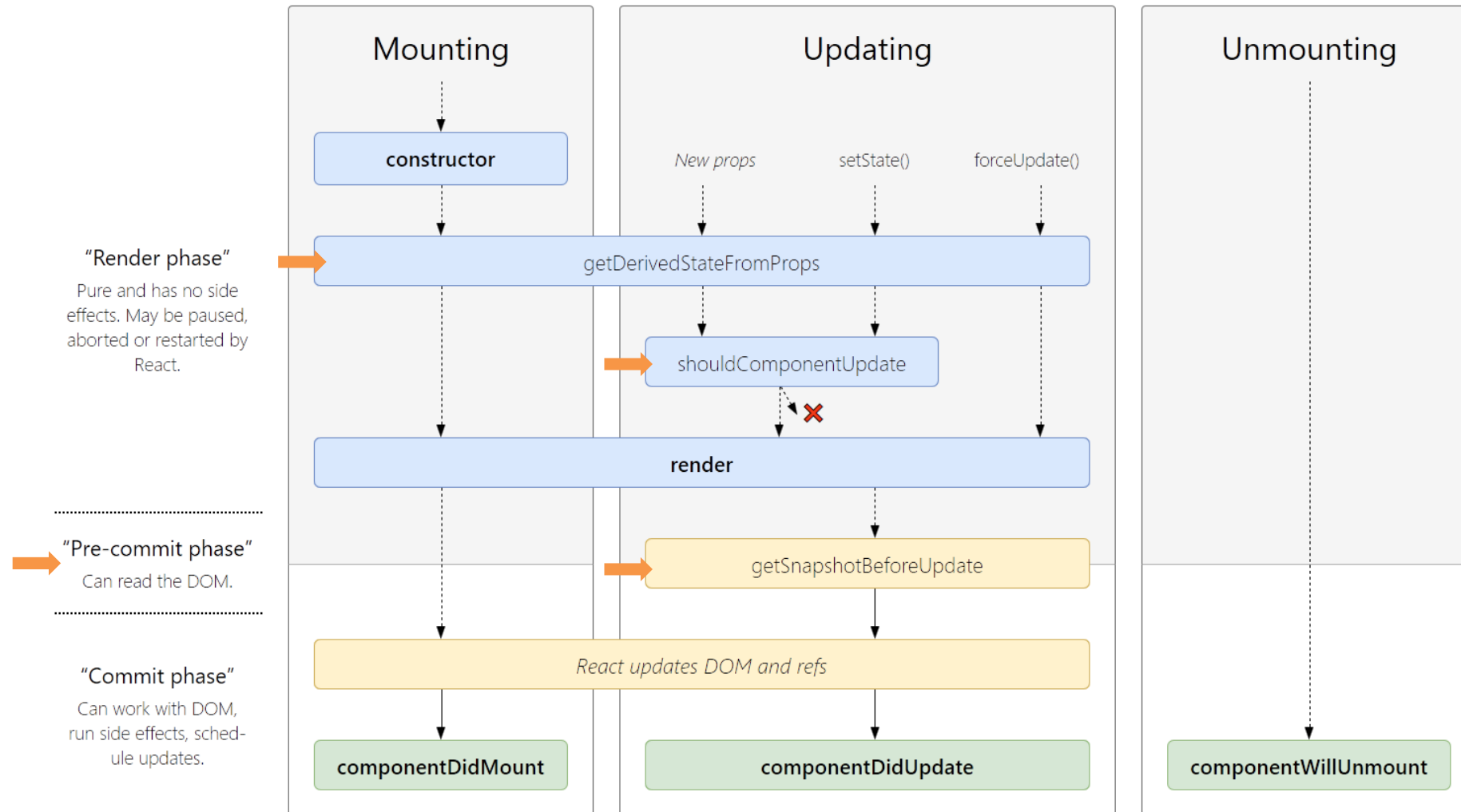
Controlling rendering

- **TL;DR:** Don't do this! Let React decide what needs to be rendered when
- `shouldComponentUpdate()`
  - If it returns false, React will skip the render() phase
    - By default, we re-render on every state change
  - Might be used for performance optimization, but may be dangerous
  - Consider also using a `PureComponent` (faster, re-renders less frequently, but must ensure string "purity" of all methods and child components)
- `forceUpdate()`
  - Forces a re-render of the component (if it depends from other data than props and state)
  - Not recommended

Error handling methods

- When errors happen, React displays an error page (with stack trace, if in development mode)

- You may catch and control errors from your child component

- A Component becomes an "Error Boundary" by implementing:
  - `getDerivedStateFromError()`
  - `componentDidCatch()`

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

https://reactjs.org/docs/error-boundaries.html

Error handling methods

- `static getDerivedStateFromError(error)`
  - Invoked after an error has been thrown by a descendant component
  - Receives the `error` that was thrown
  - Return a value to update state
    - Hoping to correct the issue
    - Rendering an alternate interface
  - Called during the "render" phase, so side-effects are not permitted

- `componentDidCatch(error, information)`
  - Invoked after an error has been thrown by a descendant component
  - Receives the `error` that was thrown
  - Receives a componentStack with `information` about the error location
  - Called during the "commit" phase, so side-effects are permitted
  - Use for logging
  - Don't call `setState`

Forms, Events and Event Handlers

FORMS IN JSX

HTML Forms

- HTML Forms are *inconsistent*: different ways of handling values, events etc. depending on the type of input element
  - Consequence of backward compatibility
- For instance:
  - onChange on a radio button is not easy to handle
  - `value` in a `textarea` does not work, etc.
- React flattens this behavior exposing (via JSX) a more uniform interface
  - Synthetic Events

Value in JSX forms

- The **value** attribute always holds the current value of the field
- The `defaultValue` attribute holds the default value that was set when the field was created
- This also applies to
  - `textarea`: the content is in the `value` attribute; it is NOT to be taken from the actual content of the `<textarea>…</textarea>` tag
  - `select`: do not use the `<option selected>` syntax, but `<select value='id'>`

Change events in JSX Forms

- React provides a more consistent **onChange** event
- By passing a function to the `onChange` attribute you can subscribe to events on form fields (every time `value` changes)
- `onChange` also fires when typing a single character into an `input` or `textarea` field
- It works consistently across fields: even `radio`, `select` and `checkbox` input fields fire a `onChange` event

Event Handlers

- An Event Handler callback function is called with one parameter: an event object
- All event objects have a standard set of properties
  - `event.target`: source of the event
- Some events, depending on categories, have more specific properties

Synthetic Events

- "High level events" wrap the corresponding DOM Events

- Same attributes as DOMEvent

- `target` points to the source of the event.

- In case of a form element
  - `target.value` = current input value
  - `target.name` = input element name

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```

Synthetic Events

Category	Events
Clipboard	onCopy onCut onPaste
Composition	onCompositionEnd onCompositionStart onCompositionUpdate
Keyboard	onKeyDown **onKeyPress** onKeyUp
Focus	**onFocus onBlur**
Form	**onChange** onInput **onInvalid** onReset **onSubmit**
Generic	onError **onLoad**
Mouse	**onClick** onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave onMouseMove onMouseOut onMouseOver onMouseUp
Pointer	onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut
Selection	onSelect
Touch	onTouchCancel onTouchEnd onTouchMove onTouchStart
UI	onScroll
Wheel	onWheel
Media	onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend onTimeUpdate onVolumeChange onWaiting
Image	onLoad onError
Animation	onAnimationStart onAnimationEnd onAnimationIteration
Transition	onTransitionEnd

Tip: Defining Event Handlers

- Define the function as a class property
  - As an arrow function
  - As a function expression, but **remember to bind it** in the constructor

```
this.handler = () => { ... }
```
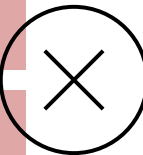✓

```
constructor(){
 this.handler = this.handler.bind(this);
}
handler () { ... }
```
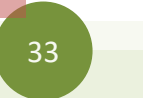✓

```
constructor(){
 this.handler = this.handler.bind(this);
}
handler = function() { ... }
```
✓

```
handler = function() { ... }
```

```
handler () { ... }
```
✗

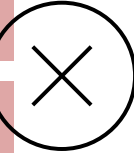# Tip: Defining Event Handlers

- Pass the *name* of the function as a prop
  - As a function object (not string)
  - Don't *call* the function

```
return <div handler={this.handler} />
```
✓

```
return <div handler={this.handler()} />
```

```
return <div handler='this.handler' />
```
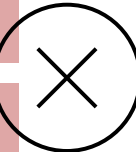✕

Tip: Defining Event Handlers

- Specify the *name* of the function prop in the event handler

- If you need to pass *parameters*, use an *arrow* function

```
return <button onClick=
    {props.handler} />
```
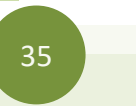✓

```
return <button onClick=
    {props.handler()} />
```

```
return <button onClick=
    {props.handler(a, b)} />
```
✗

```
return <button onClick=
    {()=>props.handler()} />
```

```
return <button onClick=
    {()=>props.handler(a, b)} />
```
✓

Who owns the state?

- Form elements are inherently stateful: they hold a value
  - Input text form, selection, etc.
- React components are designed to handle the state
- The props and state are used to render the component
  - To correctly render the component from the virtual DOM, React needs to know which value must be set in the form element
  - Hence, on every change (onChange) React must be notified to get the new value and update the component state
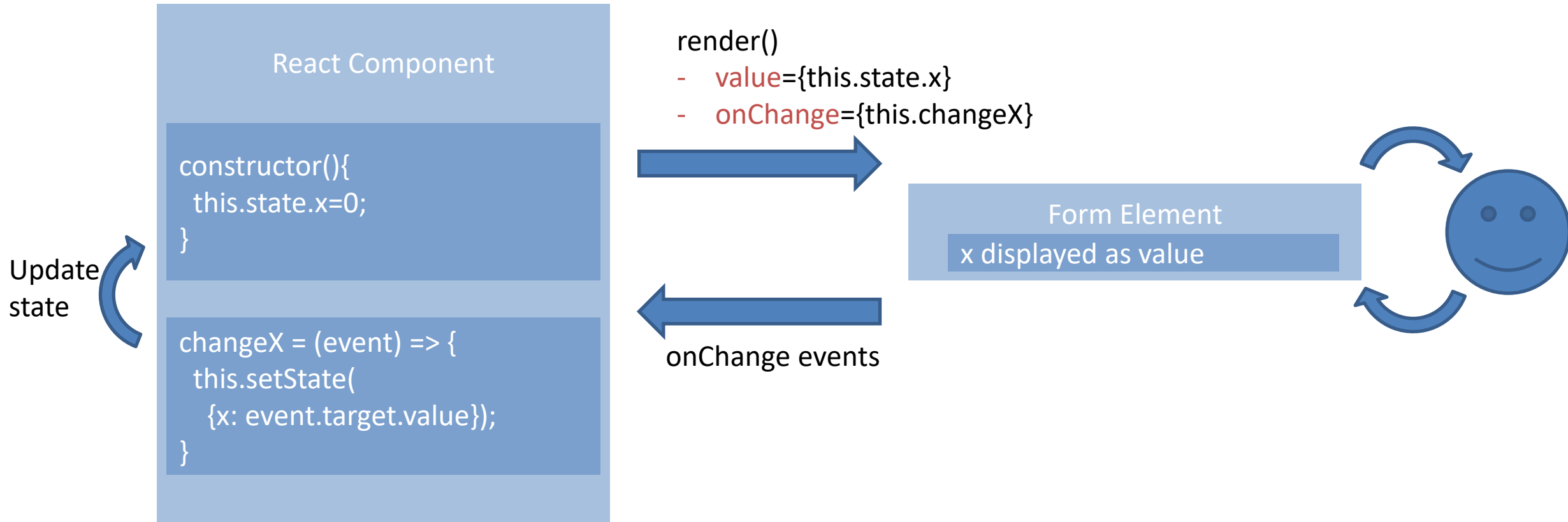
Where is the source of truth?

Controlled Components

- When the React component holds, in its state, the value to be shown in the form element, it is named a **controlled** component

Uncontrolled components

- In some occasions, it could be useful to keep the value directly in the HTML form element in the DOM: **uncontrolled** component
  - Legacy code
  - Read-only components (e.g., file selection)

Controlled components

React Component

render()
- value={this.state.x}
- onChange={this.changeX}

constructor(){
  this.state.x=0;
}

Form Element
x displayed as value

Update state

changeX = (event) => {
  this.setState(
    {x: event.target.value});
}

onChange events

Controlled component

- The event handler changes the state, `setState()` starts the update of the virtual DOM that then updates the actual DOM content

```
class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: '' };
  }
  render() {
    return <form onSubmit={this.handleSubmit}>
      <label> Name:
        <input type="text" value={this.state.value}
            onChange={this.handleChange} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  }
}
```

```
handleSubmit = (event) => {
  console.log('Name submitted: ' +
      this.state.value);
  event.preventDefault();
}


handleChange = (event) => {
  this.setState(
      {name: event.target.value}
  );
};
}
```
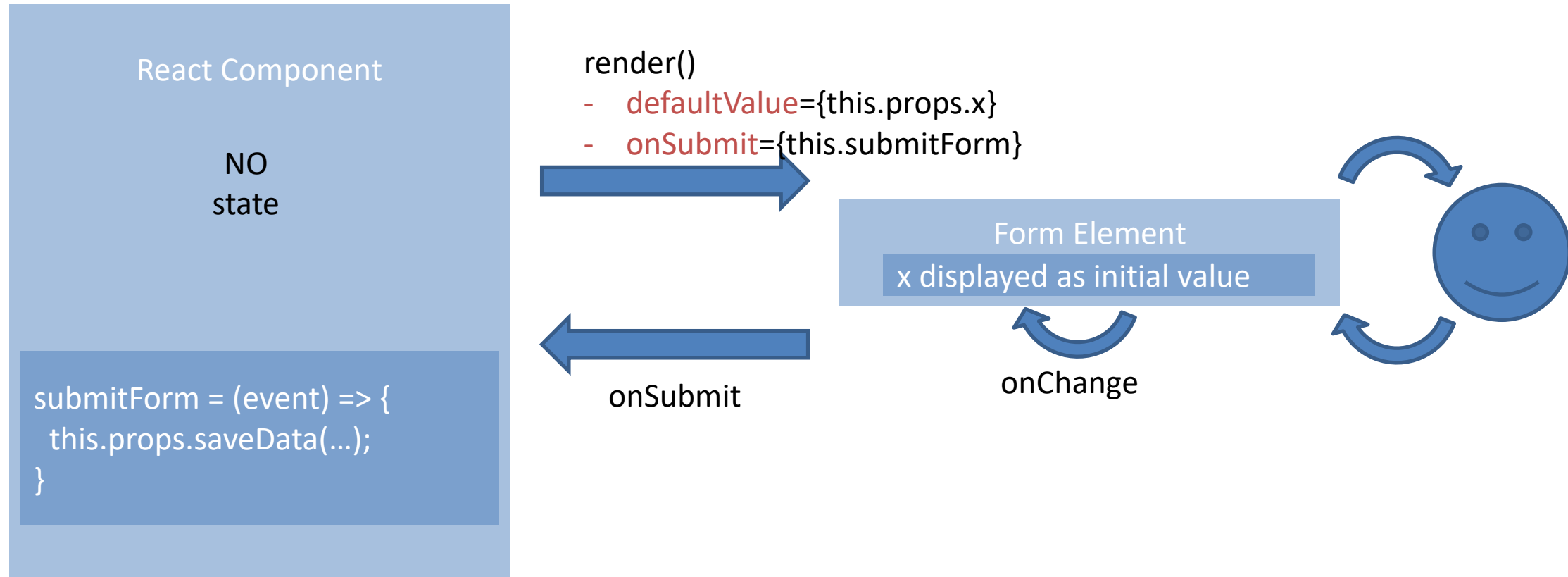
Tip: State Update for Multiple Fields

- Define only one onChange handler for all fields

- Ensure the input components have the same name as the state properties

- Use the "computed property" assignment in `setState`

- Avoids creating may independent change handlers

```
updateField = (name, value) => {
  this.setState({[name]: value});
}
```

```
render() {
  return <MyForm
    updateField={this.updateField}
    x={this.state.x} />
}
```

```
<input name='x' value={props.x}
  onChange={(ev) =>
    props.updateField(
    ev.target.name, ev.target.value)} />
```

Uncontrolled components

React Component

NO
state

render()
- defaultValue={this.props.x}
- onSubmit={this.submitForm}

Form Element
x displayed as initial value

onChange

submitForm = (event) => {
  this.props.saveData(...);
}

onSubmit

Uncontrolled component

- Create a reference from React to the DOM element. No onChange event handler required. Use defaultValue for initial value, if required (no state)

```
class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: '' };
    this.input = React.createRef();
  }
  render() {
    return <form onSubmit={this.handleSubmit}>
        <label> Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
  }
}
```

```
handleSubmit = (event) => {
  console.log('Name submitted: ' +
      this.input.current.value);
  event.preventDefault();
}
```

Controlled vs Uncontrolled

- If possible, use **controlled** components to implement forms
- In a controlled component, form data is explicitly handled by React components
- Some components do not allow set value by JS code: no alternative to uncontrolled component (e.g., file selection)

DOM Component References (ref)

- For uncontrolled components, the "current" state is in the DOM component, according to user inputs

  – How to retrieve it?

- For HTML5 forms, many validation methods and attributes already exist

  – How to re-use them, instead of re-programming everything in React?

- Some custom components (e.g., media players, map widgets, …) may be controlled through custom methods and attributes

  – How to access their functionality?

- Answer: get a *reference to the (generated) DOM Node*

https://reactjs.org/docs/refs-and-the-dom.html

44

Creating and Using Refs

- Create a local property with `createRef()`
  - `this.myRef = React.createRef()`
- Link the reference to the DOM now with `ref` attribute
  - `return <div ref={this.myRef} …/>`
- Ref is *automatically linked* when the component is mounted

- The node will be available under the `current` attribute
  - `this.myRef.current`
- All node attributes are available under current
  - `this.myRef.current.value`
- Works for DOM nodes and for class components (not function components)

https://reactjs.org/docs/refs-and-the-dom.html

Callback Refs (older method)

- You may manually link a ref using a "Callback Ref"

- Define a function on the ref attribute

- The function receives a reference to the DOM node, that can be stored in a component property

- The node itself is `this.myRef` (not `.current`)

```
<div ref={
  myRef =>
    this.myRef = myRef }>
```

Tip: Form Submission

- The `onSubmit` event is generated by the `<form>` element
- Always `event.preventDefault()` to avoid the submission (and reloading of the page)
- Perform *validation* of *all form data* before proceeding
  - Using checks on `this.state` (on a controlled component, it contains updated information) – may use `validator` https://github.com/validatorjs/validator.js
  - Using HTML5 validation attributes and methods (need a `ref` to the `<form>` to access the methods)

```
doSubmit = (item) => {
        if (this.formRef.checkValidity()) {
            this.props.sendItem(item);
        } else {
            this.formRef.reportValidity();
        }
    }
```

Alternatives to controlled components

- Sometimes, it is tedious to use controlled components
  - Need to write an event handler for every way data can change
  - Pipe all of the input state through a React component
- Alternatively, use a library such as Formik
  - Keep things organized without hiding them too much
  - Form state is inherently ephemeral and local: does not use state management solutions such as Redux/Flux which would unnecessary complicate things
  - Includes validation, keeping track of the visited fields, and handling form submission

https://jaredpalmer.com/formik

Tips: Handling Arrays in State

- React `setState()` with objects does a shallow merge of the properties
  - What happens when a property is an array? What is the correct way to handle arrays in React state?

- Use a <u>new array</u> as the value of the property

- Use a <u>callback</u> to ensure no modifications are missed

- Typical cases
  - Add items
  - Update items
  - Remove items

https://www.robinwieruch.de/react-state-array-add-update-remove

Adding Items in Array with setState()

```
// Append at the end: use concat()
// NO .push(): returns the number of elements,
not the array
...

this.state = {
    list: ['a', 'b', 'c'],
};
...

this.setState(state => {
    const list = state.list.concat(state.value);
    return { list: list };
})
```

```
// Insert value(s) at the beginning
// use spread operator

...

this.state = {
    list: ['a', 'b', 'c'],
};
...

this.setState(state => {
    const list = [newItem, ...state.list ];
    return { list: list };
})
```

shortcut:
return { list };
(Shorthand property names (ES6))

https://www.robinwieruch.de/react-state-array-add-update-remove

50

Updating Items in Array with setState()

```
// Update item: use map()
...
this.state = { list: [11, 42, 32], };
...
// i is the index of the element to update
this.setState(state => {
    const list = state.list.map((item, j) => {
      if (j === i) {
        return item + 1;  // update the item
      } else {
        return item;
      }
    });
    return { list };
});
```

https://www.robinwieruch.de/react-state-array-add-update-remove

Removing Items from Array with setState()

```
// Remove item: use filter()

...
this.state = { list: [11, 42, 32], };
...

// i is the index of the element to remove
this.setState(state => {
   const list =
      state.list.filter((item, j) => i !== j);
   return { list };
});
```

```
// Remove first item(s): use destructuring

...
this.state = { list: [11, 42, 32], };

...

this.setState(state => {
      const [first, ...list] = state.list;
      return { list };
});
```

https://www.robinwieruch.de/react-state-array-add-update-remove

Tip: Heuristics for State Lifting

- Presentational components
  - Forms, Tables, Lists, Widgets, …
  - Should contain local state to represent their display property
  - Sort order, open/collapsed, active/paused, …
  - Such state is not interesting outside the component
- Application components (or Container components)
  - Manage the information and the application logic
  - Usually don't directly generate markup, generate props or context
  - Most application state is "lifted up" to a Container
  - Centralizes the updates, single source of State truth

License