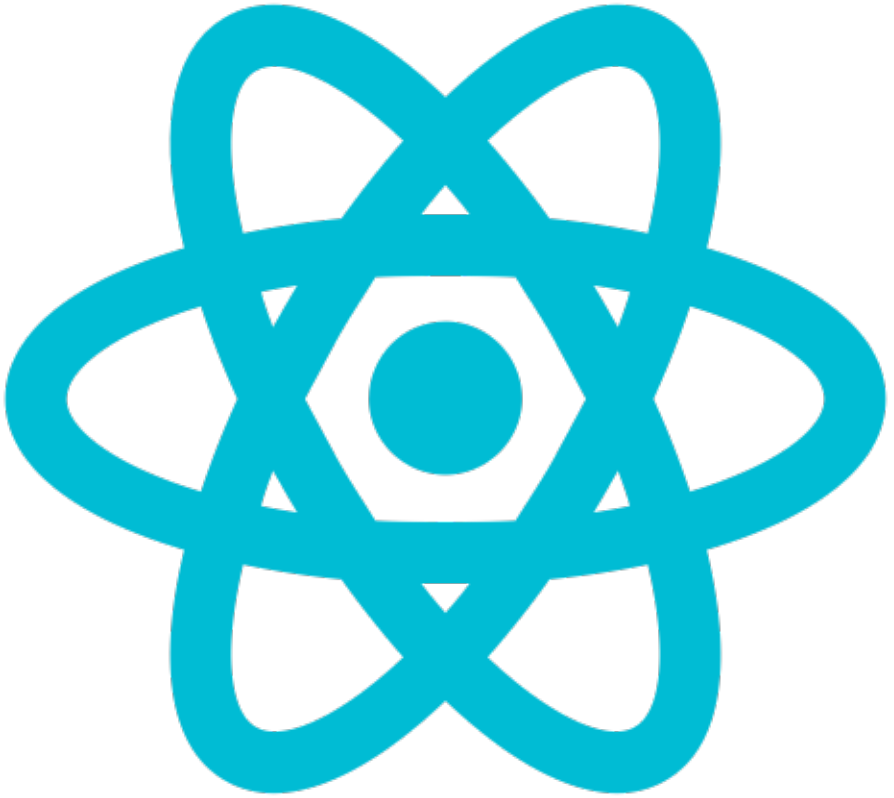
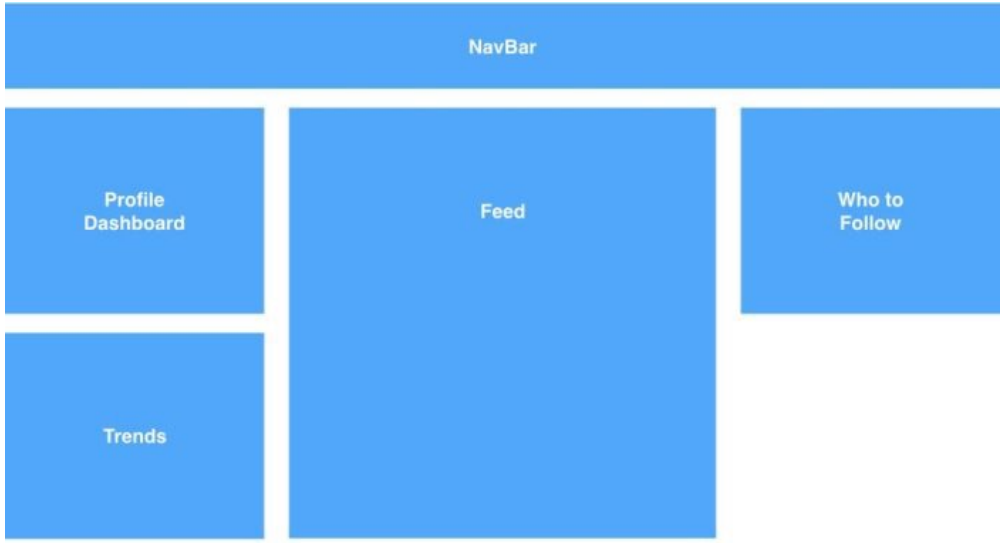
# Components and Props



# React components

- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

# Functional and Class components

- The simplest way to define a component is to write a JavaScript function:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element. We call such components “function components” because they are literally JavaScript functions.
- You can also use an ES6 class to define a component:

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

# Rendering a component

- Previously, we only encountered React elements that represent DOM tags:

```
const element = <div />;
```

- However, elements can also represent user-defined components:
- When React sees an element representing a user-defined component, it passes JSX attributes and children to this component as a single object. We call this object “props”.

```
const element = <Welcome name="Sara" />;
```

# Rendering a component (Example)

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

- Let's recap what happens in this example:

1. We call ReactDOM.render() with the <Welcome name="Sara" /> element.
2. React calls the Welcome component with {name: 'Sara'} as the props.
3. Our Welcome component returns a <h1>Hello, Sara</h1> element as the result.
4. React DOM efficiently updates the DOM to match <h1>Hello, Sara</h1>.



Try it on CodePen

# Composing Components

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```



Try it on CodePen

# Converting a Function to a Class

- You can convert a function component like Clock to a class in five steps:

1. Create an ES6 class, with the same name, that extends React.Component.
2. Add a single empty method to it called render().
3. Move the body of the function into the render() method.
4. Replace props with this.props in the render() body.
5. Delete the remaining empty function declaration.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

# Stateful vs Stateless components

- **Stateful and Stateless Components**
- Stateful and stateless components have many different names.
- *They are also known as:*
- – Container vs Presentational components
- – Smart vs Dumb components
- The literal difference is that one has state, and the other doesn't. That means the stateful components are keeping track of changing data, while stateless components print out what is given to them via props, or they always render the same thing.

# Stateful vs Stateless components (*When to use*)

- **When would you use a stateless component??**

1. When you just need to present the props
2. When you don’t need a state, or any internal variables
3. When creating element does not need to be interactive
4. When you want reusable code

- **When would you use a stateful component?**

1. When building element that accepts user input
2. ..or element that is interactive on page
3. When dependent on state for rendering, such as, fetching data before rendering
4. When dependent on any data that cannot be passed down as props

# Adding State to a component

- We will explain more about changing the state when we talk about React hooks. Let’s take a glimpse.

# Props are Read-Only

- Whether you declare a component as a function or a class, it must never modify its own props. Consider this sum function:

```
function sum(a, b) {
  return a + b;
}
```

- Such functions are called “pure” because they do not attempt to change their inputs, and always return the same result for the same inputs.
- In contrast, this function is impure because it changes its own input:

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

# Props in Functional Component

advancedHello.js

```
import React from 'react'
const advancedHello = (props) => {
   return (<h1>Hello {props.name} </h1>);
}
export default advancedHello;
```

App.js

```
function App() {
 return (
   <div className="App">
    <AdvancedHello name='Umur'></AdvancedHello>
    <AdvancedHello name='Aynur'></AdvancedHello>
   </div>
 );
}
```

# Props in Class-Based Component

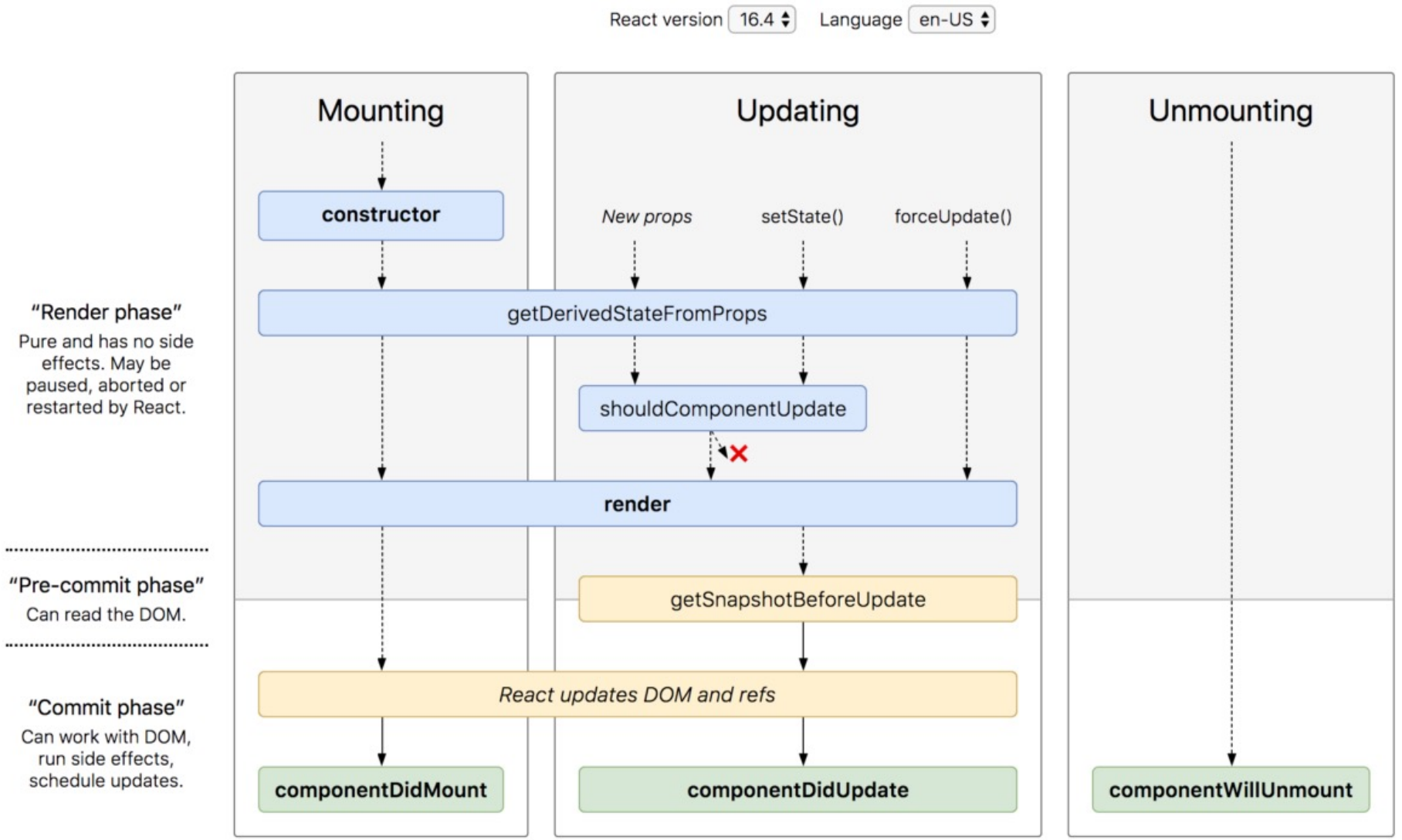
advancedGreeting.js

```
import React from 'react';

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

export default AdvancedGreeting;
```

App.js

```
function App() {
 return (
   <div className="App">
     <AdvancedGreeting name='Erol'></AdvancedGreeting>
     <AdvancedGreeting name='Ayse'></AdvancedGreeting>
   </div>
 );
}
```

# Component Lifecycle


React version 16.4
Language en-US
Mounting
Updating
Unmounting
constructor
New props
setState()
forceUpdate()
getDerivedStateFromProps
shouldComponentUpdate
render
getSnapshotBeforeUpdate
React updates DOM and refs
componentDidMount
componentDidUpdate
componentWillUnmount
"Render phase"
Pure and has no side effects. May be paused, aborted or restarted by React.
"Pre-commit phase"
Can read the DOM.
"Commit phase"
Can work with DOM, run side effects, schedule updates.

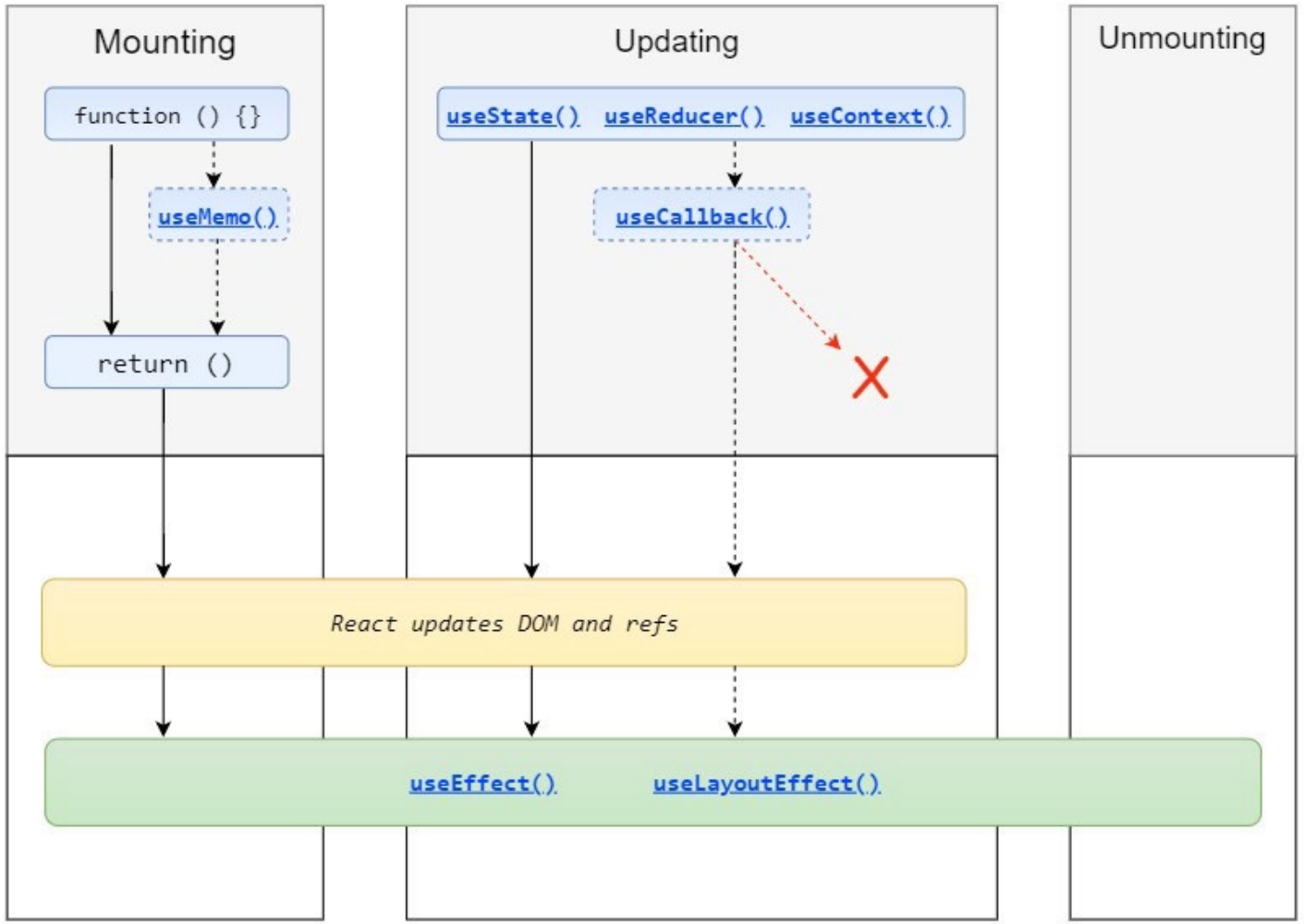
# React Hooks Lifecycle

**"Render phase"**

Pure and has no side effects. May be paused, aborted or restarted by React

**"Commit phase"**

Can work with DOM, run side effects, schedule updates.



# Handling events

- Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:
- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.
- For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

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

# Handling events

- Another difference is that you cannot return false to prevent default behavior in React. You must call preventDefault explicitly. For example, with plain HTML, to prevent the default link behavior of opening a new page, you can write:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

In React, this could instead be:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

https://www.robinwieruch.de/react-event-handler