MODULE 7
7.1 - REACT HOOKS

A FULLSTACK REACT *MASTERCLASS*
TINYHOUSE

There are two ways to create React Components:

- **Class Components**
- **Function Components**

Traditionally, React required us to create **Class Components** to:

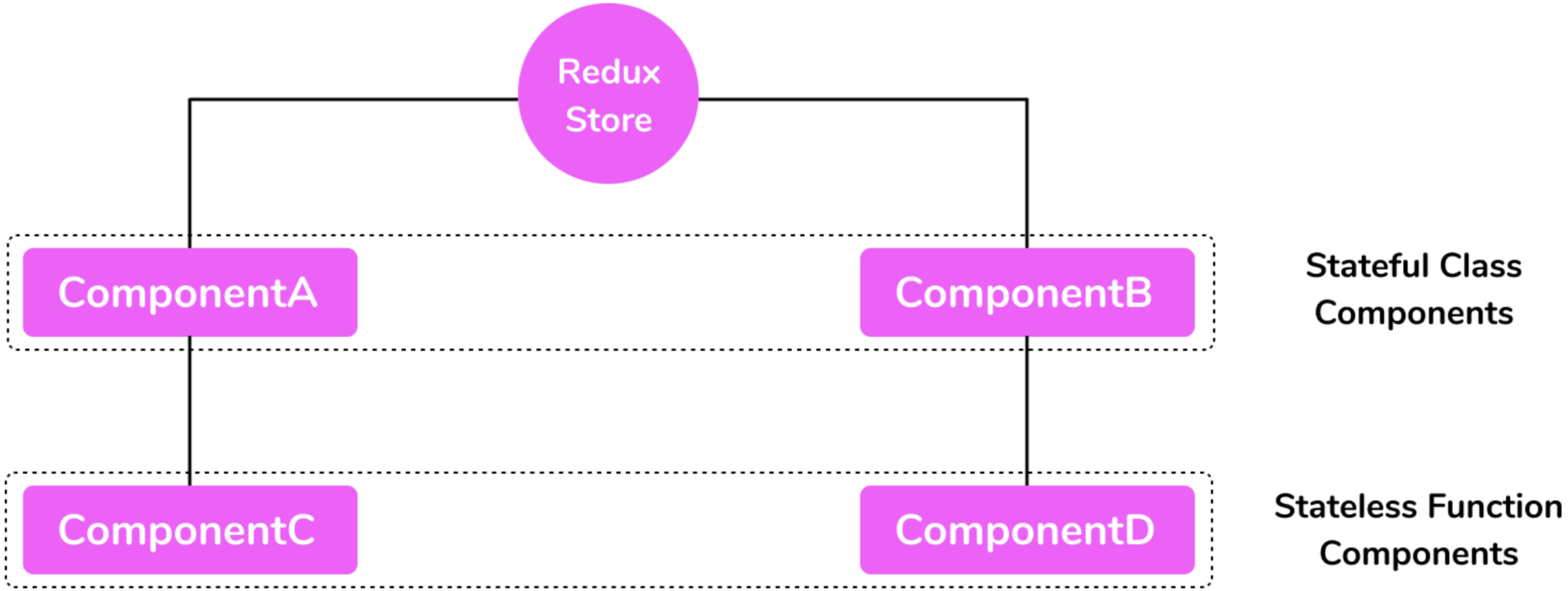
- Use state within a component
- Use other React specific features (e.g lifecycle methods)

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

class HelloWorld extends Component {
  constructor(props) {
    super(props);
    state = {
      message: 'Hello World!',
    };
  }

  componentDidMount() {} // e.g lifecycle method

  render() {
    return <h2>{this.state.message}</h2>;
  }
}
```

Function Components often appeared more "presentational"

```
import React from 'react';

const HelloWorld = ({ message, changeMessage }) => {
  return <h2 onClick={changeMessage}>{message}</h2>;
};
```

"Presentational" and "Container" Component pattern emerged from this


Redux
Store
ComponentA
ComponentB
ComponentC
ComponentD
Stateful Class
Components
Stateless Function
Components

"Presentational" and "Container" Component pattern isn't really encouraged any longer

Typical advanced patterns to handle logic reuse were:

- Higher Order Components
- Render Props

Downside is that these made components a lot more *complex*

React didn't provide a simpler/smaller primitive to adding state or lifecyle than a Class component

Motivation behind **React Hooks**

React Hooks

Allow us to handle state and other React features in Function Components

Aims to simplify re-use of *stateful* logic between components

```
import React, { Component } from "react";

class Counter extends Component {
  constructor(props) {
   super(props);
    state = {
      count: 0
    }

    this.increment = this.increment.bind(this);
    this.decrement = this.decrement.bind(this);
  }

  increment() {
    setState({
      count: this.state.count + 1
    })
  }

  decrement() {
    setState({
      count: this.state.count - 1
    })
  }

  return (
    <div>
      <p>{this.state.count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}
```

```jsx
import React, { useState } from "react";

const Counter = () => {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <p>{count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
```

Hooks allow us to *split* component logic to smaller *functions* that can be reused between components

```jsx
import React, { useState } from "react";

const useCounter = (initialState) => {
  const [count, setCount] = useState(initialState);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return [count, { increment, decrement }];
};

const Counter = () => {
  const [myCount, { increment, decrement }] = useCounter(0);

  return (
    <div>
      <p>{myCount}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
```

- Hooks are **functions**
- Hooks should always be called at the top-level of components
- Hooks should only be called from React functions
- Hooks are entirely opt-in (no breaking changes)

Great References:

- React Today and Tomorrow... | **React Conf 2018** - Link
- Hooks API Reference | **React Docs** - Link
- Rules of Hooks | **React Docs** - Link