Component Hierarchy

- When components are nested inside of other components there exists a Parent/Child relationship

```
<Parent>
    <Child />
    <Child />
    <Child />
</Parent>
```

Rendering Lists of Components

- In order to render a list components:
  - We first create a "wrapper" component that is responsible for rendering a single item of the list.
  - Then, in the Parent, we create of an array of these wrapper components and render the array.

Example

```
class ListItem extends Component {
    render() {
        return (<li>{this.props.name}</li>)
    }
}

class MyComponent extends Component {
    constructor(props) {
       super(props)
       this.data = ['File 1', 'File 2', 'File 3'];
    }

    render() {
        var toRender = [];
        this.data.forEach(function (file) {
            toRender.push(<ListItem name={file} />)
        });
        return (<ul> {toRender} </ul>);
    }
}
```

Dynamic Children

- If they are children dynamically being added/removed/shuffled around, React needs a way of uniquely identifying each child.
- For this we have the "key" attribute.
- When adding a child element to the list you will need to define the "key" attribute.

```
render() {
    var toRender = [];
    this.myObjects.forEach(function (object) {
        toRender.push(<ListItem name={object.name} key={object.id}/>)
    });
    return (<ul> {toRender} </ul>);
}
```

Exercise - Folder Exercise

- Using child components for the files recreate the folder exercise.
- This time create an array of "file" objects each with an id and fileName;
- Use the id as a key.
- eg:

```
this.files = [{
    id: '1',
    fileName: 'File1'
}, {
    id: '2',
    fileName: 'File 2'
}];
```

Accessing the children

- there is a property on the "this.props" object called "children"
- caveats:
  - if there is more than one children "this.props.children" will be an array
  - if there is only one child, "this.props.children" will be an object.
- React.children provides utilities for dealing handling "this.props.children"

React.children utility functions

- React.Children.map
  - useful for returning a modified array of children that should be rendered.
- React.Children.forEach
  - useful for iterating through an array and capturing some information from each child
- React.Children.count
  - useful for determining if "this.props.children" is an array or an object

Example

```
// Render only the elements that have a name of "Tom"
render() {
    var result = [];
    React.Children.forEach(this.props.children, function(child) {
        if (child.props.name = 'Tom') {
            result.push(child)
        }
    })
    return <div>{result}</div>
}
```

Child Mutability

- It is considered good practice **not** to manipulate children.
- Instead we clone the child and render the cloned version.
- To do this, React provides a "React.cloneElement" utility function.
- The "cloneElement" function takes an element and any additional props to apply and returns the cloned element.

Example (p1)

```
class Child extends Component {
    render() {
        return <div>Name: {this.props.childName}, Message: {this.props.message}</div>;
    }
};
```

Example (p2)

```
class Child extends Component {
    render() {
        return <div>Name: {this.props.childName}, Message: {this.props.message}</div>;
    }
};


class Parent extends Component {
    render() {
        const modfiedChildren = React.Children.map(this.props.children, function(child) {
            let toReturn = child;
            if (child.props.age < 18) {
                toReturn = React.cloneElement(child, {
                   message: 'Sorry but you do not have access to see this message'
                });
            }
            return toReturn;
        });

        return <div>{modfiedChildren}</div>
    }
};
```

Example (p3)

```
class Child extends Component {
    render() {
        return <div>Name: {this.props.childName}, Message: {this.props.message}</div>;
    }
};


class Parent extends Component {
    render() {
        const modfiedChildren = React.Children.map(this.props.children, function(child) {
            let toReturn = child;
            if (child.props.hasAccess) {
                toReturn = React.cloneElement(child, {
                   message: 'Sorry But you are not old enough to see this'
                });
            }
            return toReturn;
        });

        return <div>{modfiedChildren}</div>
    }
};

ReactDOM.render(
        <Parent>
            <Child childName="Everette" hasAccess={false} message="Super duper Message"/>
            <Child childName="Sally" hasAccess={true} message="Hi Sally How Are you"/>
        </Parent>, document.getElementById('container'))
```

Presentational vs Container Components

- React offers two ways to create components. The ones we've been working with are considered to be "Container Components", which contains its own state and is aware of the props it's receiving.
- The opposite of this is a "Presentational Component", which is not concerned with state or how it receives it's props. It's simply concerned with how things look.
- But why offer two methods? Optimization and speed! Presentational components have no state, so React looks at them as if they were just standard functions.

Let's check these out:

Presentational

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

const Title = function(props) {
  return (
    <div>{props.name}</div>
  )
}};

export default Title;
```

- no `this` keyword when accessing props
- all this component does is output a div with some text
- its only concern is to output this div, with whatever prop we feed it, nothing else

Guided Exercise - Parental Communication

The purpose of this exercise is to demonstrate how children components can communicate clicks to the parent.

- Create a Child Component that displays the "value" prop and button that can be clicked.
- Then create a Parent Component that renders a cloned version of the children, but adds a click handler prop.
- Use the React.cloneElement function to clone the child and add the click handler.

ES6 Spread operator in React

- A lot of times we want to pass through a lot of properties into children.
- But we don't want to list a ton of attributes on our elements.
- Enter: the spread operator (...)

Example

```
class ContactCard extends Component {
    render() {
        return(<div>
            <h1>First Name: {this.props.fName}</h1>
            <h1>Last Name: {this.props.lName}</h1>
            <h3>Age: {this.props.age}</h3>
            <h3>Zip Code: {this.props.zipCode}</h3>
            <h4>State: {this.props.state}</h4>
        </div>);
    }
}

function MyComp(props) {
    const aPerson = {
        fName: 'Tom',
        lName: 'Smith',
        age: '12',
        zipCode: '11231',
        state: 'CA'
    };

    return <ContactCard {...aPerson} />
}
```