Chapter 5: Props in React

Section 5.1: Introduction

`props` are used to pass data and methods from a parent component to a child component.

Interesting things about `props`

1. They are immutable.
2. They allow us to create reusable components.

Basic example

```
class Parent extends React.Component{
  doSomething(){
     console.log("Parent component");
  }
  render() {
    return <div>
         <Child
           text="This is the child number 1"
           title="Title 1"
           onClick={this.doSomething} />
         <Child
           text="This is the child number 2"
           title="Title 2"
           onClick={this.doSomething} />
       </div>
  }
}

class Child extends React.Component{
  render() {
    return <div>
       <h1>{this.props.title}</h1>
       <h2>{this.props.text}</h2>
      </div>
  }
}
```

As you can see in the example, thanks to `props` we can create reusable components.

Section 5.2: Default props

`defaultProps` allows you to set default, or fallback, values for your component `props`. `defaultProps` are useful when you call components from different views with fixed props, but in some views you need to pass different value.

Syntax

ES5

```
var MyClass = React.createClass({
  getDefaultProps: function() {
    return {
      randomObject: {},
```

```
      ...
    };
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.defaultProps = {
    randomObject: {},
    ...
}
```

ES7

```
class MyClass extends React.Component {
    static defaultProps = {
        randomObject: {},
        ...
    };
}
```

The result of `getDefaultProps()` or `defaultProps` will be cached and used to ensure that `this.props.randomObject` will have a value if it was not specified by the parent component.

Section 5.3: PropTypes

`propTypes` allows you to specify what `props` your component needs and the type they should be. Your component will work without setting `propTypes`, but it is good practice to define these as it will make your component more readable, act as documentation to other developers who are reading your component, and during development, React will warn you if you you try to set a prop which is a different type to the definition you have set for it.

Some primitive `propTypes` and commonly useable `propTypes` are -

```
  optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol
```

If you attach `isRequired` to any `propType` then that prop must be supplied while creating the instance of that component. If you don't provide the **required** `propTypes` then component instance can not be created.

Syntax

ES5

```
var MyClass = React.createClass({
  propTypes: {
    randomObject: React.PropTypes.object,
```

```
    callback: React.PropTypes.func.isRequired,
    ...
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.propTypes = {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
};
```

ES7

```
class MyClass extends React.Component {
     static propTypes = {
        randomObject: React.PropTypes.object,
        callback: React.PropTypes.func.isRequired,
        ...
    };
}
```

More complex props validation

In the same way, `PropTypes` allows you to specify more complex validation

Validating an object

```
...
    randomObject: React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    }).isRequired,
...
```

Validating on array of objects

```
...
    arrayOfObjects: React.PropTypes.arrayOf(React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    })).isRequired,
...
```

Section 5.4: Passing down props using spread operator

Instead of

```
var component = <Component foo={this.props.x} bar={this.props.y} />;
```

Where each property needs to be passed as a single prop value you could use the spread operator `...` supported for arrays in ES6 to pass down all your values. The component will now look like this.

```
var component = <Component {...props} />;
```

Remember that the properties of the object that you pass in are copied onto the component's props.

The order is important. Later attributes override previous ones.

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

Another case is that you also can use spread operator to pass only parts of props to children components, then you can use destructuring syntax from props again.

It's very useful when children components need lots of props but not want pass them one by one.

```
const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };
var component = <Component {...{foo, bar}} />;
const { foo, bar } = component.props
console.log(foo, bar); // 'foo bar'
```

Section 5.5: Props.children and component composition

The "child" components of a component are available on a special prop, `props.children`. This prop is very useful for "Compositing" components together, and can make JSX markup more intuitive or reflective of the intended final structure of the DOM:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {this.props.children}
            </div>
        </article>
    );
}
```

Which allows us to include an arbitrary number of sub-elements when using the component later:

```
var ParentComponent = function () {
    return (
        <SomeComponent heading="Amazing Article Box" >
            <p className="first"> Lots of content </p>
            <p> Or not </p>
        </SomeComponent>
    );
}
```

Props.children can also be manipulated by the component. Because props.children may or may not be an array, React provides utility functions for them as React.Children. Consider in the previous example if we had wanted to wrap each paragraph in its own **`<section>`** element:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
```

```
                {React.Children.map(this.props.children, function (child) {
                    return (
                        <section className={child.props.className}>
                            React.cloneElement(child)
                        </section>
                    );
                })}
            </div>
        </article>
    );
}
```

Note the use of React.cloneElement to remove the props from the child `<p>` tag - because props are immutable, these values cannot be changed directly. Instead, a clone without these props must be used.

Additionally, when adding elements in loops, be aware of how React reconciles children during a rerender, and strongly consider including a globally unique `key` prop on child elements added in a loop.

Section 5.6: Detecting the type of Children components

Sometimes it's really useful to know the type of child component when iterating through them. In order to iterate through the children components you can use React `Children.map` util function:

```
React.Children.map(this.props.children, (child) => {
  if (child.type === MyComponentType) {
    ...
  }
});
```

The child object exposes the `type` property which you can compare to a specific component.