1) use functional components as often as you can
2) to pass value or function to another component use props:

Parent:

```
switchNameHandler = (newName) => {

        console.log('I was clicked');
}
<Person
name= "Damon Albarn"
age= "28"
click= { this.switchNameHandler }
/>
```

Children:

```
const person = (props) => {
   return  (
      <div>
         <p onClick={props.click}> I'm {props.name} and I am {props.age} years old ! </p>
      </div>
   )
};
```

3) to bind click and past some value use **THIS**:

```
click={ this.switchNameHandler.bind(this, 'Bohdan') }
```

NOT THIS:

```
<button onClick={ () => this.switchNameHandler('Emanuello') }>Switch name</button>
```

4) use arrow functions from ES6, 'cause it's the easier way to manipulate with 'this'.
in this case, 'this' will refer to your exsiting component object
5) the best way to style - create special ComponentName.css file.
But you also can do inline-styling:
in **render**():

```
const style = {
        backgroundColor: 'white',
        font: 'inherit',
        border: '1px solid blue',
        padding: '8px',
        cursor: 'pointer'
   };
```

and **then**:

```
 <button
        onClick={ () => this.switchNameHandler('Anastasi') }
        style={style}>Switch name</button>
```

USE CAMEL CASE for css properties; NOT ->, for example - 'font-size', but 'fontSize' and etc.

6) if u want render something to DOM with condition U must use ternar operator (something ? if true : if false).

null in else condition means that nothing gonna be rendered.

Example:

```
{
    this.state.showPersons ?
    <div>
        <Person
        name={ this.state.persons[0].name }
        age={ this.state.persons[0].age } />
    </div>
     : null //render nothing
}
```

More elegant and convinion way - create jsx variable in render method and then past it into return.

Example:

```
let persons = null;
if (this.state.showPersons) {
 persons = (
  <div>
    <Person
    name={ this.state.persons[0].name }
    age={ this.state.persons[0].age } />
  </div>
);
```

and then just:

```
 return (
<div>
    {persons}
</div>
);
```

7) We must mutate state in immutable way. It means:

DON'T USE THIS:

```
const persons = this.state.persons;
```

JUST GET A COPY:

```
const persons = this.state.persons.slice();
```

or with spread operator: `const persons = [...this.state.persons];`

8) Use 'key' in list elements, because in this way we will show react witch part to rerender. Key can be any unique data, but more frequency it will be 'id'.

9) for hover and media query - good way is to use Radium - library from npm, which helps us to style over components. EVERYTHING IN REACT IS JAVASCRIPT.

Component lifecycle

10) ONLY AVAILABLE IN ***STATEFUL*** COMPONENT (not functional)

Executed during ***creation:***

1. constructor(props) - default ES6 feature. Must call super(props).
   Do: Set up State
   Don't: Cause side-effects -> reaching out web-server.
2. componentWillMount() - exist for historic reasons. ***EXISTS, BUT DO NOT USED ANYMORE.***
   Do: Update State, last-minute optimization
   Don't: Cause side-effects -> reaching out web-server.
3. render() - what it should render. How your component should like from HTML perspective. Obviously, just prepare & structure your JSX Code.
4. **After render() it renders child components.**
5. componentDidMount() - component was successfully mount.
   Do: Cause side-effects -> maybe fetch some data and so on
   Don't: update State (triggers re-render)

Update (triggered by parent) hooks:

1. componentWillReceiveProps(nextProps) - we can synchronize over local state to a props.
   Do: Sync State to Props
   Don't: Cause side-effects -> reaching out web-server.
2. shouldComponentUpdate(nextProps, nextState) - **MAY CANCEL UPDATING PROCESS.** Returning true or false (save performance and don't update).
   Do: Decide whether to Continue or Not
   Don't: Cause side-effects -> reaching out web-server.
3. componentWillUpdate(nextProps, nextState) - **EXECUTE ONLY IF** shouldComponentUpdate() **RETURNS TRUE.**
   Do: Sync State to Props
   Don't: Cause side-effects -> reaching out web-server.
4. render()
5. **Update Child Components Props**
6. componentDidUpdate()
   Do: Cause side-effects -> maybe fetch some data and so on
   Don't: update State (triggers re-render)

Update (triggered by internal change) hooks:

1. shouldComponentUpdate(nextProps, nextState) **MAY CANCEL UPDATING PROCESS.** Returning true or false (save performance and don't update).
   Do: Decide whether to Continue or Not
   Don't: Cause side-effects -> reaching out web-server.
7. componentWillUpdate(nextProps, nextState) - **EXECUTE ONLY IF** shouldComponentUpdate() **RETURNS TRUE.**
   Do: Sync State to Props

Don't: Cause side-effects -> reaching out web-server.

8. render()
9. **Update Child Components Props**
10. componentDidUpdate()
    Do: Cause side-effects -> maybe fetch some data and so on
    Don't: update State (triggers re-render)

11) **PureComponent** - React implements lifecycle hook ***shouldComponentUpdate***() without our "help". It checks if props or state are updated (different) and then decides - return true or false, it means, React decides re-render components or not by itself. Convenient way -> implement **PureComponent** in high-l evel components (containers).

12) Solving problem with unnecessary **div** in render method:

- U can return an **array** and then you don't need wrapping **div**.
  - U had this:

```
<div className={ classes.Person }>
  <p onClick={ this.props.click }> I'm {this.props.name} and
I am {this.props.age} years old ! </p>
  <p> {this.props.children} </p>
  <input type="text" onChange={ this.props.changed } value={
this.props.name }/>
</div>
```

  - Now U have this:

```
return [
      <p key="1" onClick={ this.props.click }> I'm
{this.props.name} and I am {this.props.age} years old !
</p>,
      <p key="2"> {this.props.children} </p>,
      <input key="3" type="text" onChange={
this.props.changed } value={ this.props.name }/>
       ]
```

- But convenient way is to use React 16 feature - create special component and than wrap by it all what U need.

  Aux.js

```
const aux = (props) => props.children;
export default aux;
```

  And then:

```
<Aux>
      <h2> { props.appTitle } </h2>
      <p className = {assignedClasses.join(' ')}>it 's not
me, it's you; 3</p>
      <button
```

```
            className = {btnClass}
            onClick = {props.clicked}>
            Toggle Persons </button>
        </Aux>
```

Higher order component - hoc -> is an convenient way to delete redundant **div** -> auxiliary component. With hoc components we can also return **JSON**.