Defining components

src/components/Button/Button.js

```
import React from "react";
import "./Button.css";
function Button(props) {
  return (
    <button className="Button"
      onClick={props.gotClicked}>
      {props.children}
    </button>
  )
};
export default Button;
```

Alternatively, you can use "arrow function" short-hand:

```
const Button = (props) => (
  <button className="Button"
    onClick={props.gotClicked}>
    {props.children}
  </button>
);
```

Or destructured prop parameters:

```
const Button =
    ({gotClicked, children}) => (
  <button className="Button"
    onClick={gotClicked}>
    {children}
  </button>
);
```

Useful techniques

Using `map` to loop through data

```
<div>{
  props.data.map((item, index) => (
    <p onClick={
        () => doSomething(index)}>
      {index}: {item}
    </p>
  ))
}</div>
```

Using `? :` (ternary operator) for an "if-statement"

```
<div>{
    props.image ? (
      <img src={props.image} />
    ) : <em>No image provided.</em>
}</div>
```

Class-based syntax

```
import Button from
  "./components/Button/Button.js";
function App() {
  // default state, declared as
  // object syntax
  state = {
    num: 0
  }
  // "method" syntax
  inc = () => {
    setState({
      num: this.state.num + 1,
    });
  }
  // "render" method
  render() {
    return (


      <Button
        gotClicked={this.inc}>
          Click me
          {this.statenum}
      </Button>


    )
  }
}
```

React Terms

component One discrete, re-usable, self-contained portion of React code that can be used multiple times in a project for repeatable graphical components

props Short for "properties", props are *immutable* and represent the data passed down to components from the parent of a component as attributes

class-based component Alternate option using OOP syntax

unidirectional data-flow The idea that parents pass data to children via props, while children can never interact with siblings or with their parents directly

Virtual DOM Novel technique to speed up rendering while seemingly rerenders entire page (does "dry run" to render a "virtual DOM", compares what changed with the real DOM, and only makes minimum changes)

lifting state State is best kept at the App ("top level"), and passed down to children

Array state recipes

```
// Example state:
const [arr, setArr] =
  useState(["a", "b"]);
```

Appending a new value to array:

```
let item = "c";
setArr([
  ...arr, // Include old vals
  item, // Include "item" val
]);
```

Deleting based on index:

```
let index = 1; // remove "b"
setArr([ // Update "arr"
  // Everying up until index
  ...arr.slice(0, index),
  // And everything after index
  ...arr.slice(index + 1),
]);
```

Filter: Remove from array based on condition:

```
setArr(arr.filter(
  item => item.length === 1
));
```

Object state recipes

```
const [obj, setObj] = useState(
  {name: "Kim", age: 40});
// Modifying a property of an obj
let prop = "age";
let newVal = 41;
setObj({ // Update "obj"
    ...obj, // Include other data
    skill: lang, // Modify skill
    [prop]: newVal, // Modify age
});
```

Destructuring

```
const o = {name: "jane", age: 35};
const name = o.name;
const age = o.age;
// Equivalent to
const o = {name: "jane", age: 35};
const {name, age} = o;
```

Spread

```
// Combine objects or arrays
const o = {name: "jane", age: 35};
const b = {...o, skill: "js"}
```