React 101
Codecademy
Components Interacting

Components Render other Components

Components Interact

A Component in a Render Function

We have seen that a .render() method returns an HTML-like JSX element.

Render methods can also return another kind of JSX: *component instances*.

```
class OMG extends React.Component {
  render() {
    return <h1>Whooaa!</h1>;
  }
}

class Crazy extends React.Component {
  render() {
    return <OMG />;
  }
}
```

Apply a Component in a Render Function

We know that a component can be rendered by ReactDOM.render().

What happens when a component renders another component, what happens is very similar to what happens when ReactDOM.render() renders a component.

Require A File

When we use React.js, every JS files in the application is invisible to every other JS file by default. So to render a component defined in a separate file in must first be imported.

```
import { NavBar } from './NavBar.js';
```

notice that the string at the end is different from how we have been using import.

When we use an import statement, and the string at the end begins with either a dot or a slash, then import will treat that string as a filepath. Import will follow that filepath, and import the file that it finds.

If the filepath doesn't have a file extension, then '.js' is assumed.

Note:

None of this behaviour is specific to React! Module systems of independent, importable files are a very popular way to organize code.

Difference between using {} or not using {} in import?

The way we import the content from another file will always depend on how it is being exported, for example:

```
import ComponentName from './ComponentName';
```

this means that whatever CompponentName is, it is a default export, which means that it is the only export of it's kind, this approach is mainly implemented with objects, or some classes like in the case of components.

```
class ComponentName extend React.Component {
...
}

export default ComponentName;
```

then when we see:

```
import { ComponentName } form './CompoenetName';
```

It just means that it is a named export by having the export keyword assigned to it which makes it like a property of an exported object:

```
export class ComponentName extends React.Component {
  ...
}
```

Other ways of importing:

```
import * as allComponentMethods from './ComponentName';
/* which reads: import all under the name allComponentMethods*/
```

```
import { ComponentName as NamedExported } from './ComponentName';
/*here we change the name on the name export*/
```

export

To import a variable from a file import statement isn't quite enough. We also need an export statement, written in the other file, exporting the variable we hope to grab.

export and import are meant to be used together and come from ES6's module system.

There are a few ways to use export.

Named exports:

In one file, place the keyword export immediately before something that you want to export. That something can be any top-level var, let, const, function, or class

Multiple things can be exported from a single file:

```
// Manifestos.js:

export const faveManifestos = {
  futurist: 'http://www.artype.de/Sammlung/pdf/russolo_noise.pdf',
  agile:  'https://agilemanifesto.org/iso/en/manifesto.html',
  cyborg:         'http://faculty.georgetown.edu/irvinem/theory/Haraway-
CyborgManifesto-1.pdf'
};

export const alsoRan = 'TimeCube';
```

Everything being exported can be imported in the required file as;

```
// App.js:

// Import faveManifestos and alsoRan from ./Manifestos.js:
import { faveManifestos, alsoRan } from './Manifestos';

// Use faveManifestos:
console.log(`A Cyborg Manifesto:  ${faveManifestos.cyborg}`);
```

This style of importing and exporting in JS is known as "named exports". While using named exports, you always need to wrap your imported names in curly braces.