The 7 JavaScript Skills You Need To Know For React

1. Function Declarations and Arrow Functions

The basis of any React application is the **component**. In React, components are defined with both JavaScript functions and classes. But unlike JavaScript functions, React components return JSX elements that are used to structure our application interface.

```
// JavaScript function: returns any valid JavaScript type
function javascriptFunction() {
  return "Hello world";
}

// React function component: returns JSX
function ReactComponent(props) {
  return <h1>{props.content}</h1>;
}
```

Note the different casing between the names of JavaScript functions and React function components. JavaScript functions are named in camel casing, while React function components are written with pascal casing (in which all words are capitalized).

There are two different ways to write a function in JavaScript. The traditional way, using the `function` keyword, called a **function declaration**. And as an **arrow function**, which was introduced in ES6.

Either function declarations or arrow functions can be used to write function components in React. The primary benefit of arrow functions is their succinctness. We can use several shorthands in order to write our functions to remove unnecessary boilerplate, such that we can even write it all on a single line.

```
// Function declaration syntax
function MyComponent(props) {
  return <div>{props.content}</div>;
}
```

```
// Arrow function syntax
const MyComponent = (props) => {
  return <div>{props.content}</div>;
};

// Arrow function syntax (shorthand)
const MyComponent = (props) => <div>{props.content}</div>;

/*
In the last example we are using several shorthands that arrow functions allow:

1. No parentheses around a single parameter
2. Implicit return (as compared to using the "return" keyword)
3. No curly braces for function body
*/
```

One small benefit of using function declarations over arrow functions is that you don't have to worry about problems with **hoisting**.

Due to the JavaScript behavior of hoisting, you can use multiple function components made with function declarations in a single file in whichever order you like. Function components made with arrow functions, however, cannot be ordered whichever way you like. Because JavaScript variables are hoisted, arrow function components must be declared before they are used:

```
function App() {
  return (
    <>
      {/* Valid! FunctionDeclaration is hoisted */}
      <FunctionDeclaration />
      {/* Invalid! ArrowFunction is NOT hoisted. Therefore, it must be declared befor
e it is used */}
      <ArrowFunction />
    </>
  );
}

function FunctionDeclaration() {
  return <div>Hello React!</div>;
}

function ArrowFunction() {
  return <div>Hello React, again!</div>;
}
```

Another small difference between using the function declaration syntax is that you can immediately export a component from a file using `export default` or `export` before the function is declared. You can only use the `export` keyword before arrow functions (default exports must be placed on a line below the component).

```
/*
Function declaration syntax can be immediately exported with export or export default
*/
export default function App() {
  return <div>Hello React</div>;
}

// Arrow function syntax must use export only
export const App = () => {
  return <div>Hello React</div>;
};
```

2. Template Literals

JavaScript has a clumsy history of working with strings, particularly if you want to **concatenate** or connect multiple strings together. Before the arrival of ES6, to add strings together, you needed to use the `+` operator to add each string segment to one another.

With the addition of ES6, we were given a newer form of string called a **template literal**, which consists of two back ticks `` instead of single or double quotes. Instead of having to use the+operator, we can connect strings by putting a JavaScript expressions (such as a variable), within a special `${}` syntax:

```
/*
Concatenating strings prior to ES6.
Notice the awkward space after the word Hello?
*/
function sayHello(text) {
  return 'Hello ' + text + '!';
}

sayHello('React'); // Hello React!

/*
Concatenating strings using template literals.
See how much more readable and predictable this code is?
```

```
*/
function sayHelloAgain(text) {
  return `Hello again, ${text}!`;
}

sayHelloAgain('React'); // Hello again, React!
```

What's powerful about template literals is its ability to use any JavaScript expression (that is, anything in JavaScript that resolves to a value) within the `${}` syntax.

We can even include conditional logic using the ternary operator, which is perfect for conditionally adding or removing a class or style rule to a given JSX element:

```
/* Go to react.new and paste this code in to see it work! */
import React from "react";

function App() {
  const [isRedColor, setRedColor] = React.useState(false);

  const toggleColor = () => setRedColor((prev) => !prev);

  return (
    <button
      onClick={toggleColor}
      style={{
        background: isRedColor ? "red" : "black",
        color: "white",
      }}
    >
      Button is {isRedColor ? "red" : "not red"}
    </button>
  );
}

export default App;
```

In short, template literals are great for React whenever we need to dynamically create strings. For example, when we use string values that can change in our head or body elements in our site:

```
import React from "react";
import Head from "./Head";

function Layout(props) {
  // Shows site name (i.e. Reed Barger) at end of page title
```

```
  const title = `${props.title} | Reed Barger`;

  return (
    <>
      <Head>
        <title>{title}</title>
      </Head>
      <main>{props.children}</main>
    </>
  );
}
```

3. Short Conditionals: &&, ||, Ternary Operator

Considering React is just JavaScript, it is very easy to conditionally show (or hide) JSX elements using simple if statements and sometimes switch statements.

```
import React from "react";

function App() {
  const isLoggedIn = true;

  if (isLoggedIn) {
    // Shows: Welcome back!
    return <div>Welcome back!</div>;
  }

  return <div>Who are you?</div>;
}

export default App;
```

With the help of some essential JavaScript operators, we cut down on repetition and make our code more concise. We can transform the if statement above in to the following, using the **ternary operator.** The ternary operator functions exactly the same as an if-statement, but it is shorter, is an expression (not a statement), and can be inserted within JSX:

```
import React from "react";

function App() {
  const isLoggedIn = true;

  // Shows: Welcome back!
  return isLoggedIn ? <div>Welcome back!</div> : <div>Who are you?</div>
```

```
}

export default App;
```

Ternary operators can also be used inside curly braces (again, since it is an expression):

```
import React from "react";

function App() {
  const isLoggedIn = true;

  // Shows: Welcome back!
  return <div>{isLoggedIn ? "Welcome back!" : "Who are you?"</div>;
}

export default App;
```

If we were to change the example above and only wanted to show text if the user was logged in (if `isLoggedIn` is true), this would be a great use case for the `&&` (and) operator.

If the first value (**operand**) in the conditional is true, `&&` operator displays the second operand. Otherwise it returns the first operand. And since it is **falsy** (is a value automatically converted to the boolean `false` by JavaScript), it is not rendered by JSX:

```
import React from "react";

function App() {
  const isLoggedIn = true;

  // If true: Welcome back!, if false: nothing
  return <div>{isLoggedIn && "Welcome back!"}</div>;
}

export default App;
```

Let's say that we want the reverse of what we're doing now: to only say "Who are you?" if `isLoggedIn` is false. If it's true, we won't show anything.

For this logic, we can use the `||` (or) operator. It essentially works opposite to the `&&` operator. If the first operand is true, the first (falsy) operand is returned. If the first operand is false, the second operand is returned.

```
import React from "react";

function App() {
  const isLoggedIn = true;

  // If true: nothing, if false: Who are you?
  return <div>{isLoggedIn || "Who are you?"}</div>;
}

export default App;
```

4. Three Array Methods: .map(), .filter(), .reduce()

Inserting primitive values into JSX elements is easy, just use curly braces.

We can insert any valid expressions, including variables that contain primitive values (strings, numbers, booleans, etc) as well as object properties that contain primitive values.

```
import React from "react";

function App() {
  const name = "Reed";
  const bio = {
    age: 28,
    isEnglishSpeaker: true,
  };

  return (
    <>
      <h1>{name}</h1>
      <h2>I am {bio.age} years old</h2>
      <p>Speaks English: {bio.isEnglishSpeaker}</p>
    </>
  );
}

export default App;
```

What if we have an array and we want to iterate over that array to show each array element within an individual JSX element?

For this, we can use the `**.map()**` method. It allows us to transform each element in our array in the way we specify with the inner function.

Note that it is especially concise when used in combination with an arrow function.

```
/* Note that this isn't exactly the same as the normal JavaScript .map() method, but
 is very similar. */
import React from "react";

function App() {
  const programmers = ["Reed", "John", "Jane"];

  return (
    <ul>
      {programmers.map((programmer) => (
        <li>{programmer}</li>
      ))}
    </ul>
  );
}

export default App;
```

There are other flavors of the .map() method that perform related tasks and are important to know because they can be chained in combination with one another.

Why? Because `.map()`, like many array methods returns a shallow copy of the array that it has iterated over, which enables it's returned array to be passed onto the next method in the chain.

`.filter()`, as its name indicates, allows us to filter certain elements out of our array. For example, if we wanted to remove all names of programmers that started with "J", we could do so with `.filter()`:

```
import React from "react";

function App() {
  const programmers = ["Reed", "John", "Jane"];

  return (
    <ul>
```

```
      {/* Returns 'Reed' */}
      {programmers
        .filter((programmer) => !programmer.startsWith("J"))
        .map((programmer) => (
          <li>{programmer}</li>
        ))}
    </ul>
  );
}

export default App;
```

It's important to understand that both `.map()` and `.filter()` are just different variations of the `**.reduce()**` array method, which is capable of transforming array values into virtually any data type, even non-array values.

Here's `.reduce()` performing the same operation as our `.filter()` method above:

```
import React from "react";

function App() {
  const programmers = ["Reed", "John", "Jane"];

  return (
    <ul>
      {/* Returns 'Reed' */}
      {programmers
        .reduce((acc, programmer) => {
          if (!programmer.startsWith("J")) {
            return acc.concat(programmer);
          } else {
            return acc;
          }
        }, [])
        .map((programmer) => (
          <li>{programmer}</li>
        ))}
    </ul>
  );
}

export default App;
```

5. Object Tricks: Property Shorthand, Destructuring, Spread Operator

Like arrays, objects are a data structure that you need to be comfortable with when using React.

Since objects exist for the sake of organized key-value storage, unlike arrays, you're going to need to be very comfortable accessing and manipulating object properties.

To add properties to an object as you create it, you name the property and it's corresponding value. One very simple shorthand to remember is that if the property name is the same as the value, you only have to list the property name.

This is the **object property shorthand**:

```
const name = "Reed";

const user = {
  // instead of name: name, we can use...
  name,
};

console.log(user.name); // Reed
```

The standard way to access properties from an object is using the dot notation. An even more convenient approach, however, is **object destructuring**. It allows us to extract properties as individual variables of the same name from a given object.

It looks somewhat like you're writing an object in reverse, which is what makes the process intuitive. It's much nicer to use than having to use the object name multiple times to access each time you want to grab a value from it.

```
const user = {
  name: "Reed",
  age: 28,
  isEnglishSpeaker: true,
};

// Dot property access
const name = user.name;
const age = user.age;

// Object destructuring
const { age, name, isEnglishSpeaker: knowsEnglish } = user;
// Use ':' to rename a value as you destructure it
```

```
console.log(knowsEnglish); // true
```

Now if you want to create objects from existing ones, you could list properties one-by-one, but that can get very repetitive.

Instead of copying properties manually, you can spread all of an object's properties into another object (as you create it) using the **object spread operator**:

```
const user = {
  name: "Reed",
  age: 28,
  isEnglishSpeaker: true,
};

const firstUser = {
  name: user.name,
  age: user.age,
  isEnglishSpeaker: user.isEnglishSpeaker,
};

// Copy all of user's properties into secondUser
const secondUser = {
  ...user,
};
```

What is great about the object spread is that you can spread in as many objects into a new one as you like, and you can order them like properties. But be aware that properties that come later with the same name will overwrite previous properties:

```
const user = {
  name: "Reed",
  age: 28,
};

const moreUserInfo = {
  age: 70,
  country: "USA",
};

// Copy all of user's properties into secondUser
const secondUser = {
  ...user,
```

```
  ...moreUserInfo,
  computer: "MacBook Pro",
};

console.log(secondUser);
// { name: "Reed", age: 70, country: "USA", computer: "Macbook Pro" }
```

6: Promises + Async/Await Syntax

Virtually every React application consists of **asynchronous code**–code that takes a indefinite amount of time to be executed. Particularly if you need to get or change data from an external API using browser features like the **Fetch API** or the third-party library **axios**.

Promises are used to resolve asynchronous code to make it resolve like normal, synchronous code, which we can read from top to bottom. Promises traditionally use callbacks to resolve our asynchronous code. We use the `.then()` callback to resolve successfully resolved promises, while we use the `.catch()` callback to resolve promises that respond with an error.

Here is a real example of using React to fetch data from my Github API using the Fetch API to show my profile image. The data is resolved using promises:

```
/* Go to react.new and paste this code in to see it work! */
import React from 'react';

const App = () => {
  const [avatar, setAvatar] = React.useState('');

  React.useEffect(() => {
    /*
      The first .then() let's us get JSON data from the response.
      The second .then() gets the url to my avatar and puts it in state.
    */
    fetch('https://api.github.com/users/reedbarger')
      .then(response => response.json())
      .then(data => setAvatar(data.avatar_url))
      .catch(error => console.error("Error fetching data: ", error);
  }, []);

  return (
    <img src={avatar} alt="Reed Barger" />
  );
};

export default App;
```

Instead of always needing to use callbacks to resolve our data from a promise, we can use a cleaned syntax that looks identical to synchronous code, called the **async/await syntax**.

The async and await keywords are only used with functions (normal JavaScript functions, not React function components. To use them, we need to make sure our asynchronous code is in a function prepended with the keyword `async`. Any promise's value can then be resolved by placing the keyword `await` before it.

```
/* Go to react.new and paste this code in to see it work! */
import React from "react";

const App = () => {
  const [avatar, setAvatar] = React.useState("");

  React.useEffect(() => {
    /*
    Note that because the function passed to useEffect cannot be async, we must creat
e a separate function for our promise to be resolved in (fetchAvatar)
    */
    async function fetchAvatar() {
      const response = await fetch("https://api.github.com/users/reedbarger");
      const data = await response.json();
      setAvatar(data.avatar_url);
    }

    fetchAvatar();
  }, []);

  return <img src={avatar} alt="Reed Barger" />;
};

export default App;
```

We use `.catch()` callback to handle errors within traditional promises, but how do you catch errors with async/await?

We still use `.catch()` and when we hit an error, such as when we have a response from our API that is in the 200 or 300 status range:

```
/* Go to react.new and paste this code in to see it work! */
import React from "react";

const App = () => {
```

```
  const [avatar, setAvatar] = React.useState("");

  React.useEffect(() => {
    async function fetchAvatar() {
      /* Using an invalid user to create a 404 (not found) error */
      const response = await fetch("https://api.github.com/users/reedbarge");
      if (!response.ok) {
        const message = `An error has occured: ${response.status}`;
        /* In development, you'll see this error message displayed on your screen */
        throw new Error(message);
      }
      const data = await response.json();

      setAvatar(data.avatar_url);
    }

    fetchAvatar();
  }, []);

  return <img src={avatar} alt="Reed Barger" />;
};

export default App;
```

7. ES Modules + Import / Export syntax

ES6 gave us the ability to easily share code between our own JavaScript files as well as third-party libraries using **ES modules**.

Also, when we leverage tools like Webpack, we can import assets like images and svgs, as well as CSS files and use them as dynamic values in our code.

```
/* We're bringing into our file a library (React), a png image, and CSS styles */
import React from "react";
import logo from "../img/site-logo.png";
import "../styles/app.css";

function App() {
  return (
    <div>
      Welcome!
      <img src={logo} alt="Site logo" />
    </div>
  );
}

export default App;
```

The idea behind ES modules is to be able to split up our JavaScript code into different files, to make it modular or reusable across our app.

As far as JavaScript code goes, we can import and export variables and functions. There are two ways of importing and exporting, as **named imports/exports** and as **default imports/exports.**

There can only be one thing we make a default import or export per file and we can make as many things named imports/export as we like. For example:

```
// constants.js
export const name = "Reed";

export const age = 28;

export default function getName() {
  return name;
}

// app.js
// Notice that named exports are imported between curly braces
import getName, { name, age } from "../constants.js";

console.log(name, age, getName());
```

We can also write all of the exports at the end of the file instead of next to each variable or function:

```
// constants.js
const name = "Reed";

const age = 28;

function getName() {
  return name;
}

export { name, age };
export default getName;

// app.js
import getName, { name, age } from "../constants.js";

console.log(name, age, getName());
```

You can also alias or rename what you are importing using the `as` keyword for named imports. The benefit of default exports is that they can be named to whatever you like.

```
// constants.js
const name = "Reed";

const age = 28;

function getName() {
  return name;
}

export { name, age };
export default getName;

// app.js
import getMyName, { name as myName, age as myAge } from "../constants.js";

console.log(myName, myAge, getMyName());
```